

计算机系统结构

西北电讯工程学院

苏东庄 主编

国防工业出版社



73.8712

962

计 算 机 系 统 结 构

西北电讯工程学院

苏 东 庄 主 编



国防工业出版社

1109766

内 容 简 介

本书讲述计算机系统结构的基本概念、基本原理、基本结构和分析方法。

共分八章。第一章讲述层次结构、程序的可移植性要求、应用与器件的影响。第二章讲述数据表示的确定、指令格式的优化、指令系统的分析、堆栈机器。第三章讲述重迭、流水和分布处理控制方式。第四章讲述总线、中断、通道、外围处理机。第五章讲述存贮层次、地址映象、替换算法、虚拟存贮器、Cache 存贮器。第六章讲述故障检测与定位、纠错码、可靠性模型。第七章讲述并行结构、多机系统、并行处理机、多处理机。第八章讲述文本式和图形式硬件描述语言，计算机系统和系统结构的评价。

本书可作为有关专业的教材和科技人员的参考书。

Dt01/15

计 算 机 系 统 结 构

西北电讯工程学院

苏东庄 主编

*

国防工业出版社出版

北京市书刊出版业营业许可证出字第 074 号

西北电讯工程学院印刷厂印刷

*

开本 787×1092 1/16 印张 27 1/8

印刷字数 681 千字 印数 1—11500 册

1981 年第一版 1981 年 7 月第一次印刷

统一书号：N15034(教-80) 定价：2.78 元

前 言

本书系高等院校工科电子类计算机专业统编教材之一。

本书按 1978 年全国工科计算机教材会议通过的大纲写出“讨论稿”后，除征求了有关专家的意见外，还按照 1980 年召开的、由来自全国五十二所兄弟院校的老师们参加的讨论班以及清华大学、上海交通大学、西安交通大学和国防科技大学代表参加的审稿会提出的意见进行了修改和补充。

本教材是按“研究软、硬件功能分配以及如何最佳、最合理地实现分配给硬件的功能”这个方向来编写的。目的是使读者对为什么要学习计算机系统结构以及如何学习和研究有一个明确的基本认识。本书力求着重于基本概念、基本原理、基本结构和分析方法；虽然在讲述时联系了实际机器的例子，但并不是围绕某种机型或是过多地从具体实现上的细节来讲述。此外，本书力求反映七十年代以来在系统结构上的重要进展以及八十年代的可能发展。

本课程应在“计算机（组成）原理”、“程序设计语言”和“数字逻辑”等课程之后开设。学生最好有“数据结构”方面的知识。本课程可以在“操作系统”、“编译原理”课程之后，或与它们同时开设。本书的第六、七、八章是按学生不选修“容错与诊断”、“纠错码”、“并行处理计算机结构”和“计算机系统性能评价”等课来写的。

本书是按计算机工程专业的需要来编写的，但也考虑了软件工程专业学生了解计算机系统结构的需要。在编写时还考虑了可供有关科技人员参考之用。

本书是三校合写的。清华大学金兰教授写了第七章；华东师范大学张东韩付教授、唐培艺同志写了第六章；西北电讯工程学院苏东庄付教授写了第一章，张志华同志写了第二、三章，李学干、张志华同志写了第四章，李学干同志写了第五章，马玉珍同志写了第八章。苏东庄付教授还对第二、三、四、五、八章进行了修改和补充，并对全书进行了统编。李学干同志承担了除第七章之外的全书文稿整理工作。

本书的主审单位是清华大学，主审人是房家国付教授和薛宏熙同志，他们对本书的编写始终给予了积极的支持和帮助；参加审稿会的所有代表都提出了很好的修改意见。我们对他们表示衷心的感谢。参加讨论班的老师们提出了很多宝贵意见，我们也对他们表示衷心的感谢。

本书的编写和出版工作得到了西北电讯工程学院教材处和图书馆的大力支持。有关领导部门、四机部教育局以及研究所和兄弟院校的很多领导和同志们对本书的编写给予了热情的鼓励，并提出了宝贵意见。我们在此表示热忱地感谢。

因时间紧迫，本书各章的习题这次来不及附上。

本教材除个别章外，还缺乏教学实践的检验，定会有不少缺点和错误，恳切希望读者予以指正。

编 者

1981 年

目 录

前 言

第一章 计算机系统结构的基本概念

§ 1 计算机系统与计算机系统结构	1
1.1 计算机系统的组成	1
1.1-1 由软件和硬件组成	1
1.1-2 计算机系统的多级层次结构	2
1.2 计算机系统结构的定义	6
1.2-1 从程序设计者看	6
1.2-2 从计算机设计者看	8
1.2-3 要结合起来全面看	8
1.2-4 器件、硬件、软件功能分配的演变	11
1.3 计算机的分型与结构的关系	12
1.3-1 按什么分型	13
1.3-2 系统结构与分型	15
§ 2 计算机系统结构发展的回顾	16
2.1 Von Neumann 型机器的结构特点	16
2.2 计算机系统结构的某些进展	17
2.2-1 运算器与运算方法	1
2.2-2 通道与 I/O 处理机	17
2.2-3 微程序	20
§ 3 软件与应用对系统结构的影响	22
3.1 程序的可移植性要求对结构的影响	23
3.1-1 采用统一的高级语言	23
3.1-2 系列机概念	24
3.1-3 模拟与仿真	30
3.2 应用对系统结构的影响	33
3.2-1 多功能通用机概念	34
3.2-2 吸收专用机系统结构的成果	35
§ 4 器件的发展对系统结构的影响	37
4.1 器件发展的简要回顾	37
4.1-1 器件性能价格比的指数上升	37
4.1-2 非用户片、现场片与用户片	38
4.2 器件的发展对逻辑设计方法的影	

响 41

4.3 器件的发展是推动系统结构前进的重要因素 42

主要参考文献

第二章 指令与编址

§ 1 数据表示	46
1.1 设计系统结构首先要确定数据表示	46
1.1-1 数据结构与数据表示	46
1.1-2 数据表示的确定	50
1.2 浮点数基值的选择和下溢处理	52
1.2-1 浮点数基值的选择	52
1.2-2 浮点数的下溢处理	57
1.3 自定义数据表示与向量数据表示	59
1.3-1 自定义数据表示	59
1.3-2 向量数据表示	64
§ 2 地址形成	66
2.1 逻辑地址空间与物理地址空间	66
2.2 有效地址形成举例	70
§ 3 指令系统	72
3.1 指令格式的优化	72
3.1-1 Huffman 压缩概念	73
3.1-2 操作码与指令字的优化表示	75
3.2 指令系统的分析	81
3.2-1 IBM 370 指令系统简介	81
3.2-2 指令系统的改进途径	85
3.2-3 指令系统的发展	95
§ 4 堆栈机器的结构特点	102
4.1 堆栈数据结构及其实现	102
4.2 算术表达式的求解	104
4.2-1 逆波兰表示式	104
4.2-2 堆栈机器的运算型指令和栈顶寄存器	105
4.3 程序的调用与链接	108
4.3-1 堆栈结构是实现程序调用的有效工具	108

4.3-2 程序调用指令和返回指令	110
4.4 堆栈型机器与通用寄存器机器的简 单比较	112

主要参考文献

第三章 控制方式

§1 重迭方式	116
1.1 顺序解释方式	116
1.2 重迭解释方式	117
1.3 DJS-240 机的相关处理	119
1.3-1 指令相关的处理	119
1.3-2 主存空间数相关的处理	119
1.3-3 通用寄存器组的相关处理	120
§2 流水方式	123
2.1 基本概念	123
2.1-1 从重迭到流水	123
2.1-2 流水结构的分类	124
2.2 主要性能及其分析	128
2.2-1 吞吐量	128
2.2-2 效率	129
2.2-3 实例分析	131
2.3 相关处理和控制机构	133
2.3-1 局部性相关的处理	133
2.3-2 全局性相关的处理	136
2.3-3 流水机器的中断处理	139
2.4 向量的流水处理	140
§3 分布处理方式简述	144
3.1 引言	144
3.2 智能终端简介	146
3.3 局部式分布处理系统结构简介	148

主要参考文献

第四章 输入输出系统

§1 输入输出系统的发展过程	152
§2 总线结构	156
2.1 总线的类型	157
2.1-1 专用总线	157

2.1-2 非专用总线	158
2.2 总线的控制方式	160
2.2-1 集中式总线控制	160
2.2-2 分布式总线控制	162
2.3 总线的通讯技术	164
2.3-1 同步通讯	164
2.3-2 异步通讯	164
2.4 数据宽度与总线线数	166
2.4-1 数据宽度	166
2.4-2 总线的线数	167
§3 中断系统	168
3.1 基本概念	168
3.2 中断的分类和分级	170
3.3 中断系统的软硬功能分配	174
§4 通道	176
4.1 基本概念	176
4.1-1 通道的分类	176
4.1-2 通道流量的分析	180
4.1-3 通道系统的功能	183
4.1-4 接口总线	184
4.2 通道结构及其操作过程	186
4.2-1 输入输出指令	186
4.2-2 通道指令和通道程序	189
4.2-3 输入输出中断	191
4.2-4 通道状态字	192
4.2-5 通道的工作过程举例	192
§5 外围处理机	197
5.1 基本概念	197
5.2 CDC-CYBER170 外围处理机 I/O 系统	199
5.2-1 概述	199
5.2-2 指令系统	199
5.2-3 CYBER 170 外围处理机的特 点	200
5.3 通道结构 I/O 系统与外围处理机结 构 I/O 系统的简单比较	203

主要参考文献

第五章 存贮体系

§1 引言	206
1.1 存贮技术: 容量、速度与价格的矛盾	206
1.1-1 存贮器的基本要求	206
1.1-2 主存和辅存	207
1.1-3 价格问题	210
1.1-4 主存容量、速度与CPU速度的关系	212
1.2 存贮体系原理	214
1.2-1 存贮体系的形成与发展	214
1.2-2 存贮体系的基本要求和性能评价	218
1.3 并行主存系统	221
1.3-1 并行访问多字	221
1.3-2 m体并行交叉存取	222
1.3-3 模m的大小与转移概率 λ 及吞吐量的关系	224
1.3-4 多端存贮器	227
1.4 相联存贮器	227
§2 程序的局部性与定位	229
2.1 程序的局部性和工作区	229
2.2 程序的定位	231
2.3 分段与分页	232
2.3-1 段式	233
2.3-2 页式	235
2.3-3 段式和页式的比较	236
2.3-4 段页式	237
§3 地址的映象与变换	239
3.1 全相联映象及其变换	240
3.2 直接映象及其变换	242
3.3 组相联映象及其变换	243
3.4 段相联映象	246
3.5 对标志表的分析	246
3.6 散列(Hashing)概念在地址变换中的应用	247
§4 替换算法及其实现	249
4.1 替换算法的分析	249
4.1-1 几种替换算法	249
4.1-2 堆栈型替换算法的特点	252
4.2 替换算法的实现	254

4.2-1 使用位法	254
4.2-2 堆栈法	256
4.2-3 比较对法	257
§5 虚拟存贮器	259
5.1 虚拟存贮器原理	259
5.1-1 虚地址到辅存实地址的变换	259
5.1-2 多用户虚拟存贮器	260
5.1-3 虚拟存贮器工作的全过程	262
5.1-4 快表与慢表	264
5.2 影响虚拟存贮器某些指标的因素	268
5.2-1 主存空间利用率	269
5.2-2 主存的命中率	270
§6 Cache 存贮器	272
6.1 基本结构	272
6.2 影响“Cache-主存”层次性能的因素	275
§7 主存保护与主存控制部件	278
7.1 主存保护	278
7.2 主存控制部件	281

主要参考文献

第六章 可靠性技术

§1 基本概念	283
1.1 故障的类别及产生的原因	283
1.2 冗余技术	285
§2 故障诊断	289
2.1 故障定位测试法	289
2.1-1 D算法	291
2.1-2 布尔差分法	294
2.2 微诊断法	295
§3 误差校正码	297
3.1 线性分组码	298
3.1-1 线性分组码及其生成矩阵	298
3.1-2 距离、重量和纠错能力	299
3.1-3 一致校验矩阵与伴随式	301
3.1-4 海明码与推广海明码	303
3.1 循环冗余码在磁带机中的应用	305
§4 可靠性模型和分析	309
4.1 可靠性模型	309
4.2 可靠性分析的例子	311
4.3 冗余结构的可靠性分析	314
4.3-1 备件替换冗余系统	314

4.3-2 N模冗余系统	315
4.3-3 混合冗余结构	316
§5 容错技术应用举例	318

主要参考文献

第七章 多机系统

§1 计算机系统结构中并行性的发展和多机系统类型	322
1.1 计算机系统结构中并行性概念的发展	322
1.1-1 并行性的广义理解	322
1.1-2 计算机系统结构向并行处理系统发展的途径和趋势	323
1.2 多机系统的特性	325
1.2-1 多机系统的耦合度	325
1.2-2 多处理机的特性和优点	326
1.3 多机系统的分类	327
1.3-1 按并行性等级分类	327
1.3-2 与其它分类法的比较	329
§2 并行处理机	330
2.1 并行处理机的特点与组成	330
2.1-1 并行处理机的工作原理和组成	330
2.1-2 并行处理机的专用性特点	331
2.2 阵列处理机的结构	332
2.2-1 ILLIAC IV的结构原理	332
2.2-2 处理单元	333
2.2-3 控制部件	334
2.2-4 阵列存储器	335
2.3 阵列处理机的算法举例	335
2.3-1 有限差分问题	336
2.3-2 矩阵问题	336
2.3-3 累加和	338
2.4 并行处理机的近期发展	339
2.4-1 阵列处理机的评价	349
2.4-2 MPP 位平面阵列处理机	340
2.4-3 BSP 科学处理机	342
2.5 SIMD 计算机的互连网络	345
2.5-1 互连网络问题的重要性	345
2.5-2 单级互连网络	345
2.5-3 循环互连网络和多级互连网络	348

§3 多处理机	352
3.1 多处理机与并行处理机的区别	353
3.2 多处理机硬件系统结构	353
3.2-1 总线结构	354
3.2-2 交叉开关结构	356
3.2-3 多端口存储器结构	357
3.2-4 开关枢纽结构	357
3.3 程序并行性	359
3.3-1 算术表达式的并行运算	359
3.3-2 递归程序的并行性	361
3.3-3 程序并行性的分析	364
3.4 并行进程的控制和调度	366
3.4-1 并行任务的派生与汇合	366
3.4-2 同步与互斥	369
3.4-3 资源分配和进程调度	371

主要参考文献

第八章 描述与评价

§1 硬件描述语言	377
1.1 什么是硬件描述语言	377
1.2 计算机设计语言 CDL	379
1.2-1 CDL 描述符	379
1.2-2 用 CDL 描述一台计算机	383
1.2-3 用 CDL 描述一台微程序计算机	388
1.2-4 CDL 语言在设计数字系统中的应用	392
1.2-5 模拟测试	394
1.3 交互式计算机图形语言	399
§2 性能评价	407
2.1 前言	407
2.2 评价计算机系统的步骤	410
2.3 计算机系统的性能指标	411
2.4 评价研究的分类	414
2.5 评价技术简述	415
2.6 对系统结构的评价	416
2.6-1 评分法	416
2.6-2 典型程序法	422

主要参考文献

第一章 计算机系统结构的基本概念

本章首先讲述计算机系统的组成和本书所用的计算机系统结构的定义，同时叙述系统结构与组成、实现、分型等的关系；而后简述系统结构近三十年来的进展，并在此基础上分析应用与器件对系统结构的影响。以便在学习后面各章之前，能对计算机系统结构有一个基本的了解。

§ 1 计算机系统与计算机系统结构

本节先讨论计算机系统的组成，而后着重讲述什么是计算机系统结构及其在构成计算机系统中的作用。最后，分析计算机的分型（例如划分成巨型机、大型机、中型机、小型机、微型机等）与计算机系统结构的关系。

1.1 计算机系统的组成

1.1-1 由软件和硬件组成

大家知道，计算机系统是由硬件和软件组成的。硬件是计算机系统在实际装置，从使用者来看，它主要指的是机器指令系统以及中央处理机、存储器、外部设备和它们之间的信息联结。软件则是计算机系统中用户共同使用的一组程序（由机器语言构成）和有关的资料，它变得日益复杂和完善，是使用计算机系统所必须的。软件一般指的是汇编程序、编译程序、操作系统、调试程序、数据库管理系统以及各种应用程序包等。它的作用或是为了便于计算机系统的使用、或是用于提高计算机系统的效率、或是用于扩展硬件的功能等。

一方面，软件随着计算机系统使用等的需要而不断扩大。明显的例子是操作系统的日益庞大，因为使用者日益希望只需发出少数几条命令，就能使计算机系统执行复杂的动作。另一方面，软件和硬件的分界面又是模糊不清，而且是动态地在不断变化着。例如，早期的机器没有乘、除法指令，因此乘、除法运算是通过子程序（软件）实现；后来的机器则把这部分软件“硬化”，即用硬件来实现乘、除法运算。就是到最近，同一种运算或操作（例如浮点运算、字符行运算等）在微型机和一般小型机中是用软件来实现的；而在大、中型机中则是用硬件来实现。又如，一般机器只有整数（定点数）、实数（浮点数）和逻辑数的数据硬件表示，因而复杂的数据结构，如向量、数组等只能用软件实现；但目前已有一些机器把这些软件硬化，使机器具有向量或数组硬件表示及相应的向量与数组运算。其它，如中断处理、存贮管理等软、硬功能分工也是随不同时期、不同机器而动态地在改变着，这个特点是学习本课程时要始终注意的。

由上可见，软件和硬件在逻辑功能上是等效的。由软件实现的操作（如编译操作、操作系统的基本操作等）在原理上是可以硬化成由硬件来实现；同样，由硬件实现的操作（如某条机器指令所完成的操作，条件码的置定等等）在原理上也是可以用软件的模拟来实现。由

自动机的基本理论可知，只要机器有“相减”及“转移”这两种指令就可用于算题，求解。这样，具有相同功能的计算机系统，其软、硬件功能分配是可以在很宽的范围内变化，如图1.1所示。选择什么样的分配比例，主要应取决于在现有硬件状况（主要是逻辑和存贮的器件状况）下的性能价格比。提高硬件功能的比例可以提高速度，减少所需的存贮容量，但会提高成本，以及降低硬件的利用率和计算机系统的灵活性和适应性；相反，提高软件功能的比例可以降低造价，提高灵活性和适应性，但速度要下降，所需的存贮器容量要增加。目

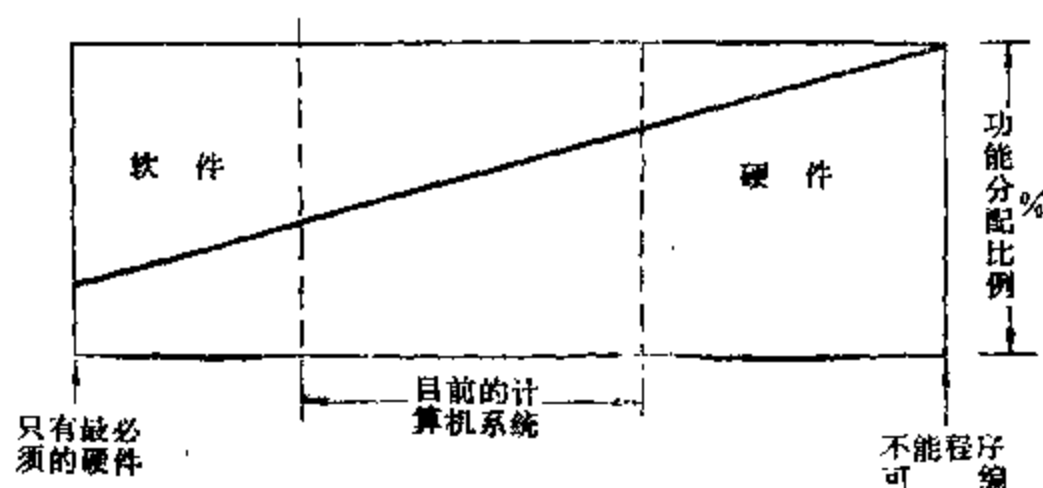


图 1.1 计算机系统的软、硬件功能分配

前，计算机系统的软、硬件功能分配对现有硬件状况、软件算法和解题算法是适应的。但这决不是一成不变的，尤其是在八十年代硬件价格随着超大规模集成电路技术的发展而日益下降的时候更是如此。

那么，对于由紧密相关的硬件和软件组成的计算机系统，我们可以用什么样的观点，从整体上去认识和分析它呢？一种观点是把计算机系统看成是按功能划分的多级层次结构。这和软件的发展过程是一致的。

1.1-2 计算机系统的多级层次结构

大家知道，早先的机器语言（即机器指令系统）由于受电子器件数量限制只能是最基本和简单的，而且使用者必须编写出用二进制表示的程序，这当然很不方便，很不适应于解题的需要及计算机使用范围的扩大。因此，在五十年代先是出现了符号式程序设计语言（汇编语言）。对它，程序员只需用诸如 ADD(加)、SUB(减)、DIV(除)和 MUL(乘)等符号来编制程序，而不必用加、减、乘、除指令所对应的二进制操作码来编制；而且程序员还可以用符号名称，而不必用二进制代码去指明指令或数据的地址。尽管汇编语言程序的每个语句和机器指令基本上仍是一一对应，但却方便多了。这样，对程序员来讲，可以把汇编语言看成是改进了的机器语言，只是没有实际的机器硬件与它直接对应，它的实现是经汇编程序翻译成真正存在的机器语言。我们可以把这想象成是在实际机器级（其机器语言由硬件实现）之上出现新的“虚拟”机器级，其机器语言为汇编语言，如图1.2所示。整个汇编语言（L2）程序（源程序）先经汇编程序变换成等效的机器语言（L1）程序（目标程序），而后再在实际机器级执行目标程序以获得结果。这样一种先把源程序变换成目标程序，而后再在机器上执行目标程序以获得结果的技术称为翻译。

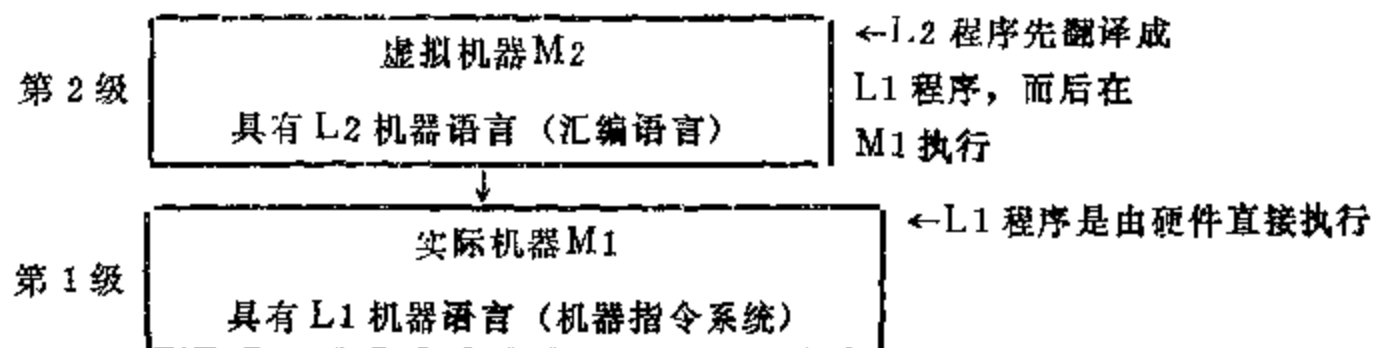


图 1.2 汇编虚拟机的实现

由于汇编语言的语法、语义结构仍然和机器语言的基本一样,而且和解题所需的仍然差别较大,因此紧接着汇编语言,又出现了面向题目的高级语言,如 FORTRAN, ALGOL 等。同理,我们可以设想成在汇编语言级之上又出现了高级语言级,它的实现是先经编译程序把高级语言程序翻译成汇编语言程序(或是机器语言程序),而后再逐级的实现,如图 1.3 所示。程序员在用高级语言编制程序时就如同面对着其机器语言为高级语言的这样一种虚拟机。

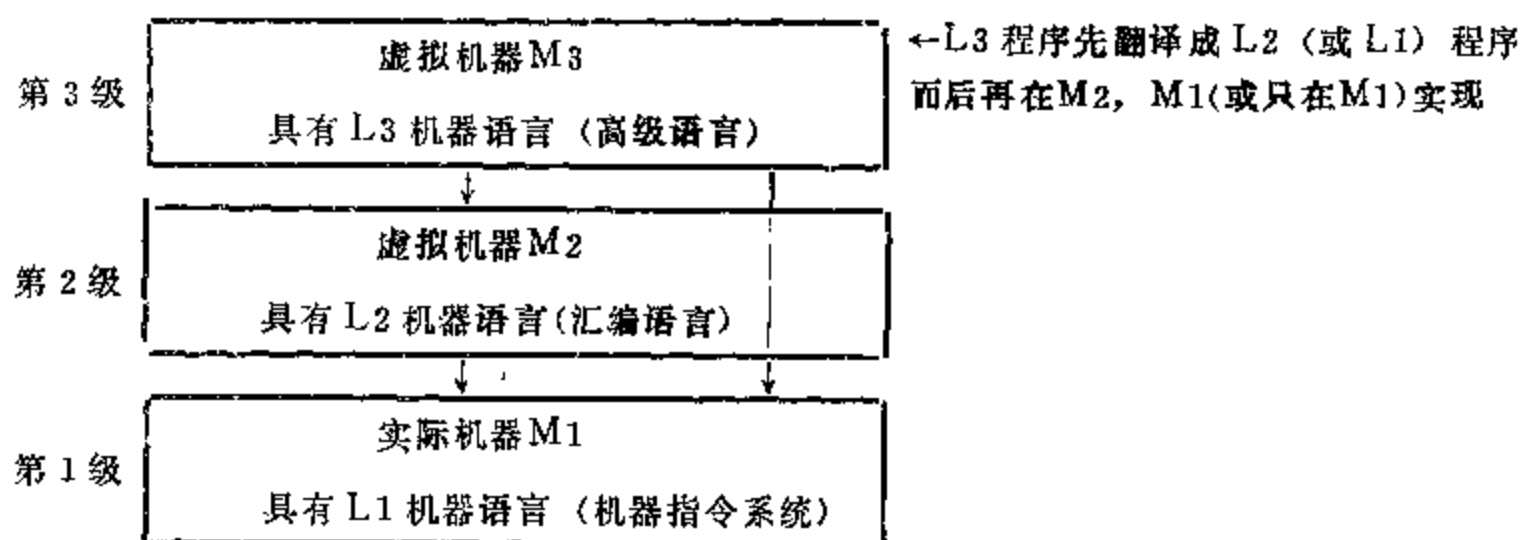


图 1.3 高级语言虚拟机的实现

这种虚拟机的垂直式层次概念对于理解各种语言的实质及其实现是有好处的。高级语言程序先翻译再执行的这样一种实现技术虽然已经持续了廿多年,但它是立足于当时的硬件条件(指硬件状况只适合提供简单的机器语言)。显然,随着我们对程序设计语言和编译技术认识的深入以及超大规模集成电路技术的发展和运用,我们应该探索并一定能够找到实现高级语言的更好办法。

这种层次概念还可引伸、应用于机器内部。对于采用微程序控制的机器,大家知道,每条机器指令对应一串微指令(一段微程序),而每条机器指令的实现是通过这一串微指令的执行。这样,我们又可以把图 1.2 的实际机器级分解成如图 1.4 所示的二级层次结构。微指令所执行的是最基本的操作,如寄存器间的数据传送、数据经过加法器或乘法器、主存与寄存器间的数据传送等等。但是,在这里并不是如同汇编语言或高级语言程序那样,先把高一

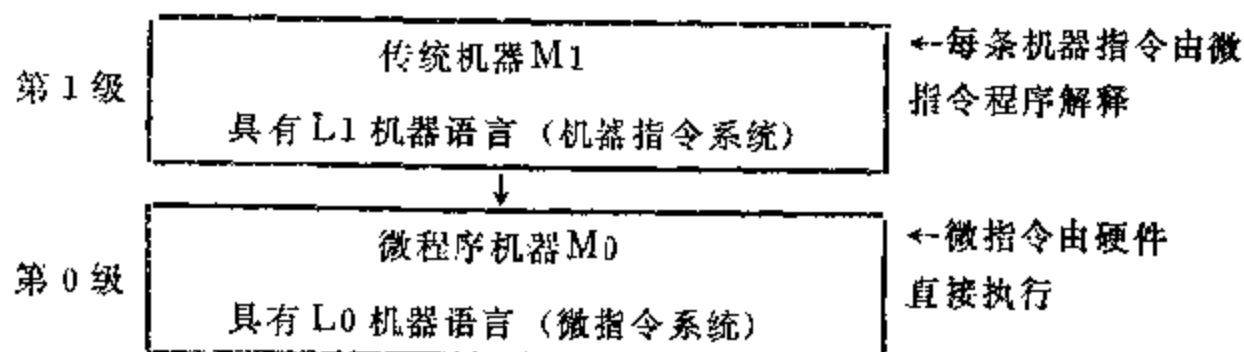


图 1.4 机器指令由微指令程序解释

级机器语言的程序翻译成低一级语言的等效程序,然后再执行它;而是每条机器指令由一串微指令实现,实现完某条机器指令后再由程序内的下条机器指令控制,进入实现它的另一串微指令。这种实现过程称为解释。如图 1.5 所示。这个过程也可理解为在 M_i 机器 (这里是微程序机器 M_0) 上,用一串 M_i 机器指令 (这里是微指令) 的执行来仿真比它高级的机器的一条语句 (这里是比 M_0 高一级的 M_1 机器指令)。用微程序解释机器指令意味着把软件技术引入到传统机器的内部,这对提高硬件的规整性以适应大规模集成电路技术的需要是很有益处的。

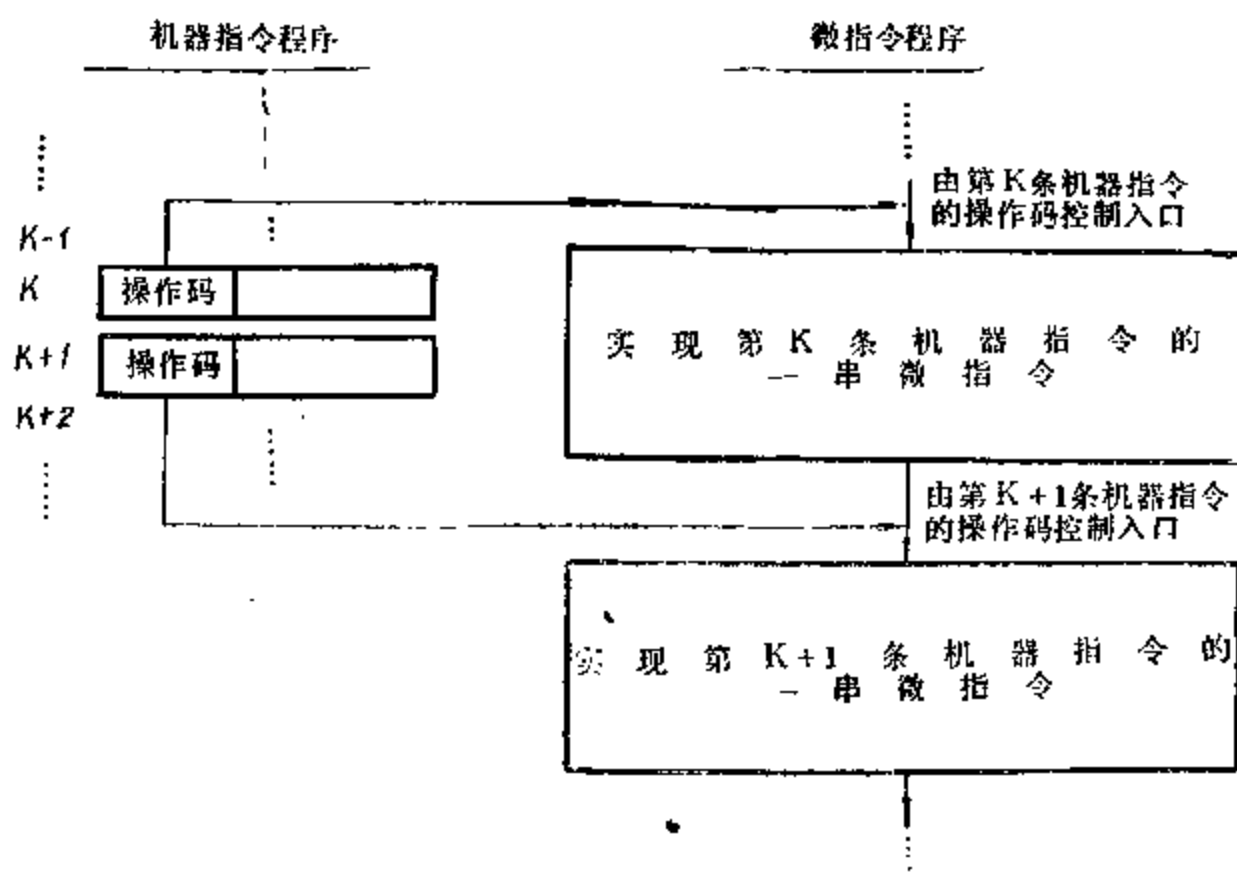


图 1.5 用微指令程序解释机器指令

翻译和解释是语言实现的二种基本技术,对于微程序控制的机器,其高级语言的实现,这二种技术都一定要用到,即先把高级语言程序经编译程序翻译成机器 (传统的) 语言程序,而后再经微程序对每条机器指令进行解释来实现。还要注意到,由高级语言程序变换成机器语言程序的过程中也不是全部只采用翻译技术,而往往是翻译和解释这二种技术并用。一般是先把高级语言源程序翻译成易于执行的某种中间型式,而后再用机器语言程序解释它 (译码及执行)。至于机器语言程序的每条机器指令,那又需经微程序解释。

上面分析了可以把计算机系统看成由高级语言虚拟机器、汇编语言虚拟机器、传统机器语言机器和微程序语言机器所构成的多级层次结构。那么,操作系统在这个层次结构中应处于什么位置呢? 这个问题比较复杂。

大家知道,操作系统实质上是传统机器的引伸,它提供了传统机器所没有的,但为汇编语言和高级语言的使用和实现所需的某些基本操作和数据结构,如文件结构与文件管理的基本操作、存贮体系以及多道程序和多重处理所用的某些操作等等。这些操作 (可以看成是操作系统的指令系统或称为作业控制语言) 和数据结构在已有的机器上大多是经机器语言程序的解释来实现。从上述分析看,操作系统级应是位于传统机器级之上,汇编语言机器级之下。当然,有些机器语言指令,如某些具体的 I/O 操作指令是要被操作系统“挡”住的,但大部分机器语言指令,如运算类指令等等,却是原样不动地穿过操作系统这级,而也包括在操

作系统级的指令系统之内。

但是，从另一方面看，操作系统还具有提高计算机系统的效率以及去控制汇编、高级等语言的实现和作业的运行等的功能；从这点看，似乎又应该把操作系统置于前述按语言结构划分的层次结构之外，并让它能作用于几乎所有各级。

另外，目前已有一些机器，尤其是今后的机器，它们的操作系统不是用汇编语言编写，而是用高级语言（不是用的面向解题的算法语言，而是用的面向系统软件的高级语言）编写。这样，操作系统级似乎又应该在高级语言机器级之上。

总之，操作系统级的位置是不能简单地指明的，不过从操作系统的基本功能来看，把它置于传统机器级与汇编语言之间基本上是能反映出其主要作用的，因此，用图 1.6 的多级层次结构来认识包括操作系统的计算机系统还是适宜的。图中每级对应一类机器（各有其自己的机器语言）。在这里，“机器”被定义为能存贮和执行程序的算法和数据结构的集合体。各

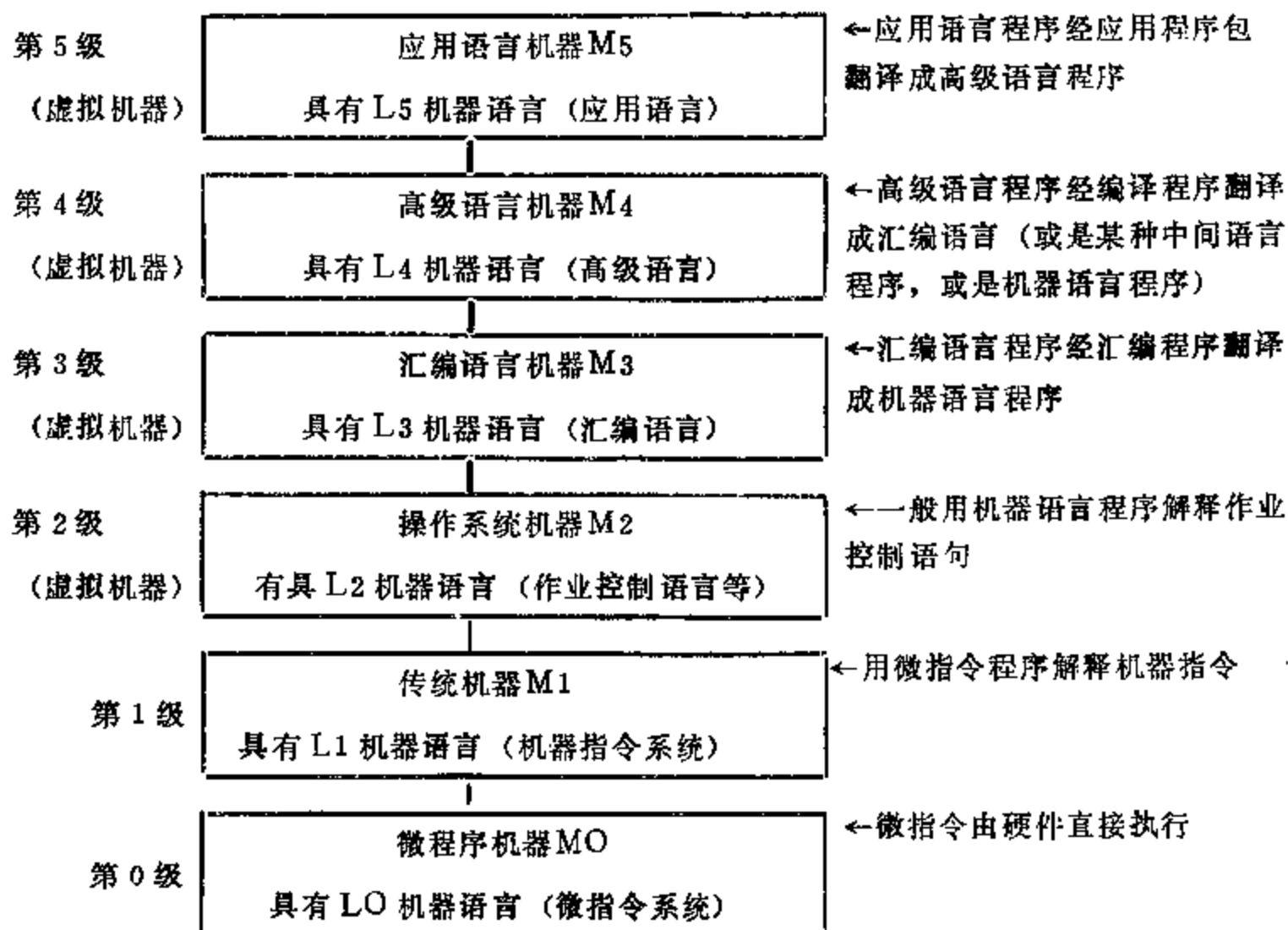


图 1.6 计算机系统的多级层次结构

级机器的算法和数据结构的实现方法不同；M0 是硬件实现；M1 是微程序（固件）实现；M2 到 M5 是软件实现。我们称由软件实现的机器为虚拟机器以区别于由硬件或固件实现的实际机器。要注意，机器的实现和题目的得到解答不是一回事，后者是要从你用的那级开始逐级变换直至到达最低级，经硬件实现才能得到；而前者主要表现为如何把该级的程序或是翻译成比它低一级的语言的程序，或是由低一级的程序所解释。因此，相邻级的语言的语法结构差别不要过大，以便于翻译或解释。当然，有的级的程序可能翻译成更低级的程序，例如高级语言程序可能直接翻译成机器语言程序；有的级的语言可能被更低级的程序所解释，例如操作系统的某些命令就可能由比它低二级的微程序解释。操作系统机器的这种例子表明虚拟机器也不一定非得全由软件实现，其中有些操作也可能是固件或硬件实现。循此，为什么

就不能设想直接由微程序或硬件实现高级语言机器或是用固件实现操作系统机器呢？采用何种实现方式，要从整个计算机系统的效率、速度、造价、资源的使用状况等等方面全面考虑。而且要软件、硬件（包括固件）综合平衡，这点一定要认识到。对计算机系统的使用者，除了关心他直接接触的那级机器（或是应用语言机器、或是高级语言机器）的状况外，当然还得关心他所选用的计算机系统是否能提供更好的性能价格比。

对于不采用微程序控制的机器（从目前趋势看主要是大型机和巨型机，因为目前的微程序控制满足不了它们的速度要求），则只是没有第0级，而是直接由硬件实现机器指令系统。除此之外，上述关于计算机系统层次结构的分析对它仍然适用。

也可以考虑在第0级之下再分级。例如有的微程序机器采用二级微程序（称为微程序与毫微程序）去解释机器指令。每条微指令不是直接由硬件执行，而是由比它低一级的毫微程序解释。另外，也可考虑把第0级的硬件再往下分级，如分成部件级、线路级、器件级等，但对于大规模集成电路技术普遍采用的八十年代来说，这种分级的意义是不大的。

上面我们分析了计算机系统是如何由硬件和软件组成的。那么，本课程所要讲述的“计算机系统结构”在计算机系统的构成中又是起着什么样的作用呢？

1.2 计算机系统结构的定义

计算机系统结构（Computer Architecture）这个名词从七十年代以来已被广泛使用，但至今还没有统一的定义，或是说还没有能够确切地定义。这和前述软件和硬件的分界面模糊不清而且在不断变化着以及器件集成度的迅速发展都是有关的。

本书中，计算机系统结构（以下有时简称计算机结构）不是指的计算机系统的结构，而是指计算机的系统结构。

结合§1.1关于计算机系统多级层次结构的分析，可以看出，对计算机的结构，从不同的级看，从各个级的使用者看，其含义、概念和着重点是不同的。“从程序设计者看的计算机结构”和“从计算机设计者看的计算机结构”这二者在概念上差别最大。下面先分别从这两方面进行分析。

1.2-1 从程序设计者看

Amdahl等人于1964年在介绍IBM360时提出，计算机结构是从程序员所看到的计算机的属性，即其概念性结构与功能特性。

这个说法至今仍被不少人所接受，但它并不确切，还需进一步分析。因为按计算机系统的层次概念，从不同级的程序员看，计算机是具有不同的属性。

例如，从使用高级语言（如FORTRAN）的程序员看，NOVA机和PDP-11机，DJS-130机和DJS-180机就几乎没有什么差别，即具有几乎相同的属性；或是说这些机器之间本来存在的差别是高级语言程序员所“看不见”的，所不需要知道的。这种本来是存在的事物或属性，但从某种角度看好象不存在的概念称为透明性概念，它对于理解软、硬件之间的界面有好处。当然，对同一种高级语言，各个厂家实质上都是有些差异的，各有其“方言”及某些不同的规定，一般做不到使某个型号上运行的高级语言程序毫不修改地能在另一型号的机器上运行。而且，在编制高级语言程序时，往往还要考虑所用机器的字长及主存容量

等。但就这些机器的概念性结构及功能特性，却是高级语言程序员所看不见的，即对高级语言程序员是透明的。图 1.7 说明了这点。

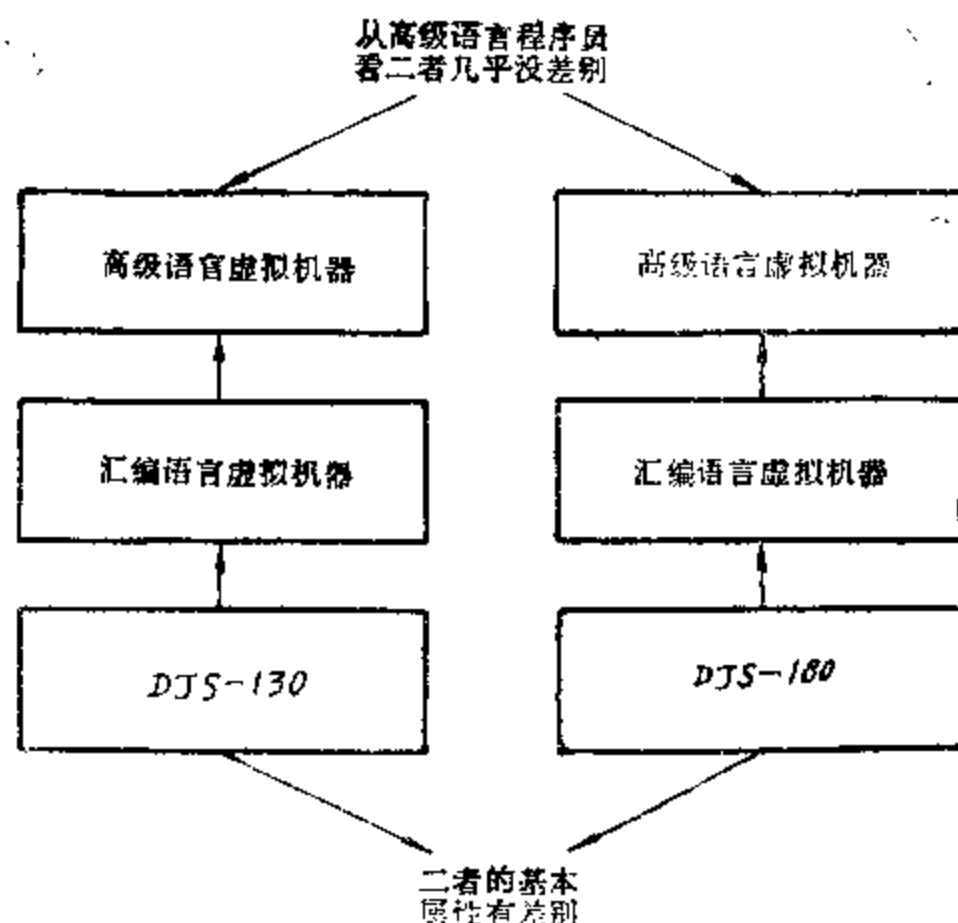


图 1.7 从高级语言程序员看的计算机属性

然而，从机器语言程序员或汇编语言程序员看，上述二种计算机的属性却是不同的，主要表现为指令系统（如操作类型和编址方式）和输入/输出设备连接方式的不同。Amdahl 等人当时指的计算机结构就是从这个角度看的计算机的概念性结构和功能特性。

从机器（汇编）语言程序员（也可以包括编译程序的设计者）看，具有相同计算机结构的机器，其机器（汇编）语言程序（也可以包括编译程序）可以通用，而不必顾及各个机器是用什么方法实现其系统结构的。但它们的操作系统却应有差别，以利于发挥具体硬件的结构特点。

一般说，从程序设计者看的计算机结构指的是由机器语言程序员（包括汇编语言程序员和编译程序设计者）所看到的传统机器级（参看图 1.6）的属性。它是程序员为编制出能在机

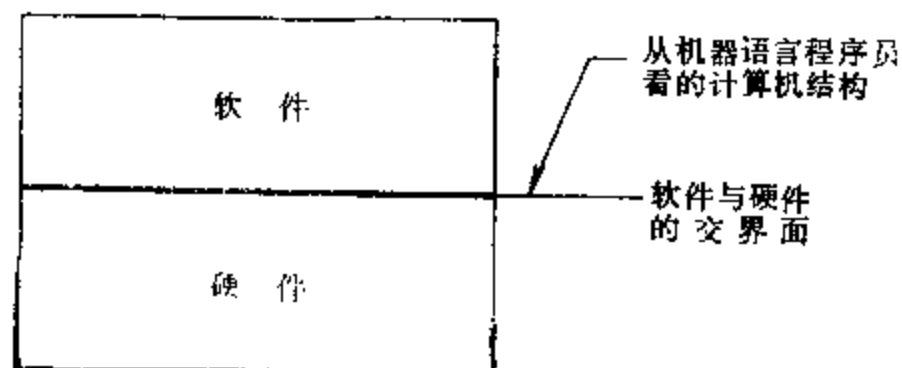


图 1.8 计算机结构是软/硬件交界面

器上正确运行的机器语言程序所必须了解的计算机结构。按此定义的计算机结构实质上指的是软件与硬件的交界面（图 1.8）。它的设计者要考虑如何使程序能被机器所识别，如何给程序所用数据编址以及如何表示数据等等；他要定义机器指令的操作类型和格式、数据的类型和格式；还要确定指令间的排序方式和机构，存储器的最

小编址单元、编址方式和保护的方法，以及输入、输出设备如何与机器接口等等。

按上述定义，DJS-130 机与 DJS-180 机，NOVA 机与 PDP-11 机当然是有不同的计算

机结构。

还要注意，“从程序设计者看”应该是即包括“从系统程序（如编译程序和操作系统等）设计者看”也包括“从应用程序设计者看”这二个方面。计算机系统结构的设计者一定要从系统程序 and 应用程序的要求来设计机器。当然，计算机结构的全面定义还应包括下面要讲到的，从计算机设计者看的计算机结构。

1.2-2 从计算机设计者看

从计算机设计者看，对计算机结构的研究应是在现有器件及设备的基础上，如何最佳、最合理地把诸如加法器、计数器、寄存器、堆栈、存贮器、磁盘机、磁带机、输入/输出设备等部件组成计算机，以实现程序设计者所提的设计要求。“最佳、最合理”的主要指标是速度、可靠性与价格。它并不着眼于如加法器、计数器、存贮器和外部设备控制器等的具体的逻辑设计；也不包括具体的工程实现，如信号传输、插件装配、机架结构与散热、通风等，虽然它们对机器的实现，尤其是巨型机的实现起着某种关键性的作用。

从计算机设计者看，计算机结构这三十年来主要进展是围绕如何提高速度。例如，重迭结构（如输入、输出和运算的重迭、同时执行等）、流水线结构（如浮点运算中的求阶差、对阶、尾加和规格化的流水同时进行等）就是为适应于提高速度的要求而提出来的，这些在下面还要分析。

有些人为了要区分从程序员看和从计算机设计者看的计算机结构，而把前者称为计算机结构，后者称为计算机组成。即计算机结构指的是从程序员看的计算机的概念性结构和功能特性，而计算机组成指的则是这种计算机结构的实现。例如，指令系统属计算机结构；而指令的实现，如取指令、取操作数、运算、送结果等的实现则属计算机组成。这样，具有相同结构（如指令系统）的计算机，其组成却可由于诸如速度要求不同等各种因素而有差异。例如有的机器其指令实现中的取指令、取操作数、运算、送结果等是顺序地执行的，而有的机器为提高速度却使这四个操作重迭地执行等等。按此，从计算机设计者来看，可以认为计算机结构与计算机组成是同义的，在目前的文献资料中这二者有时就是混用的，另外有的人把计算机的组成称为计算机的实现，并进而划分为逻辑实现和物理实现二级。逻辑实现表现为大家所熟悉的逻辑框图，而物理实现则主要表现为所用的装配技术和器件工艺。

1.2-3 要结合起来全面看

本来，正确的途径应是既从程序员角度，也从计算机设计者角度，结合起来设计计算机系统结构，然而，这二者却往往是脱节的。这三十年来，计算机设计者对计算机结构的不少改进往往没给程序的设计带来直接的好处；而程序设计者所看到的计算机结构这二十年来却几乎没有什么变化。也就是说，软件设计者是立足于几乎二十年前的计算机结构来设计日趋复杂的软件。这就使得软件设计所需工时膨胀到了惊人的程度，设计费用昂贵，设计、调试周期很长，维护复杂。有的人甚至说，目前操作系统已膨大到了顶点。另外，软件的运行是要占机器时间的。目前真正用于解题的时间往往占不到机器运行时间的一半，在中央处理机，解题时间所占比例往往还要小。这种把复杂麻烦的负担尽量扔给软件的传统做法是有其历史根源的。因为早期的计算机所用电子器件既昂贵又不可靠，体积又大，当时只有尽可能

地使计算机结构简单到必不可少的地步。但是，到了逻辑器件体积已缩小到几百分之一，存储器件已缩小到几千分之一，价格已下降几百倍的今天，仍然维持当时的软、硬件功能分配概念，那不仅不合理，而且已越来越成为计算机系统进一步发展的障碍。

下面我们结合计算机系统的设计方法来进一步分析。计算机系统的设计方法可以有“由上往下”和“由下往上”二种思路。

参看图 1.6 关于计算机系统的多级层次结构，所谓“由上往下”是指的先确定面对使用者那级虚拟机器的基本特性，如基本命令、指令或语言结构、数据的类型和格式等，而后再逐级往下设计，如图 1.9 所示。当然，每一步的设计还应包括各级的实现（例如，是采用翻译到下一级语言还是采用下一级语言解释等），而且每级的设计还要考虑如何使上一级能优化实现。这样设计出来的计算机系统，对于设计时所面向的这种应用其效能必然是很好的。这就是所谓“专用机”的概念，不过这不是指的过去那种只用（或主要用）硬件来实现某种特定函数的“专用机”。然而，当应用对象或应用范围改变时，这样设计出来的计算机系统，其软、硬件就会不适应，甚至会使整个计算机系统的效率大大下降。更何况软、硬件，尤其是软件的研制周期有时需要几年。因此，当软、硬件都配好时，其应用要求往往又会改变。所以，各级都按应用要求优化设计的情况是很少见的，尤其是硬件，它要求尽可能

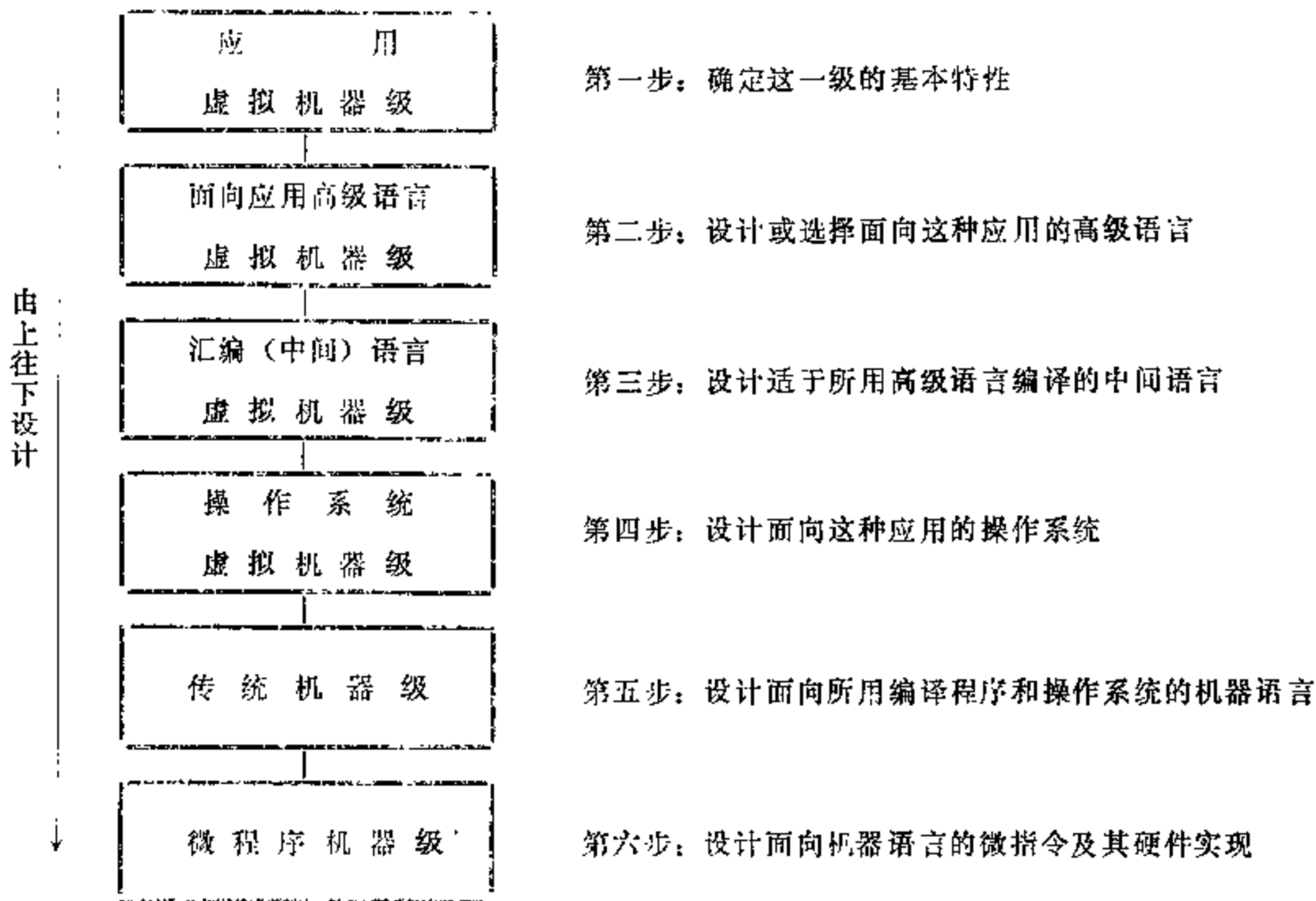


图 1.9 计算机系统由上往下的设计方法

大批量生产，生产厂应避免生产批量少的、专门面向某种应用的专用系统。这样，就是采用“由上往下”的这种设计方法，其传统机器级或微程序机器级也一般不是专门设计，而是在可能买得到的各种机器中“选型”，这当然就难真正面向应用来优化实现。

另一种是与此相反的“由下往上”的设计方法。就是根据当时能拿得到的器件状况，参照或吸收已有各种机器的优点，首先把微程序机器级（如果采用微程序控制）及传统机器级研制出来，而后再在已有硬件基础上逐步配操作系统和编译程序等软件。至于计算机系统的使用者则根据所提供的语言类型和数据型式采用相适应的算法以实现应用的要求。这种方法如图 1.10 所示。当然，操作系统的研制和编译程序的研制可以并行进行。由于要适应多种用户的需要，其操作系统往往也有多种，如各种分时操作系统及实时操作系统等；其高级语言则有 FORTRAN、ALGOL、COBOL、BASIC、APL、PL/I 及 PASCAL 等等。不少计算机系统的设计就是采用这种设计方法。然而这种方法容易造成软、硬件的脱节。其一是软件的

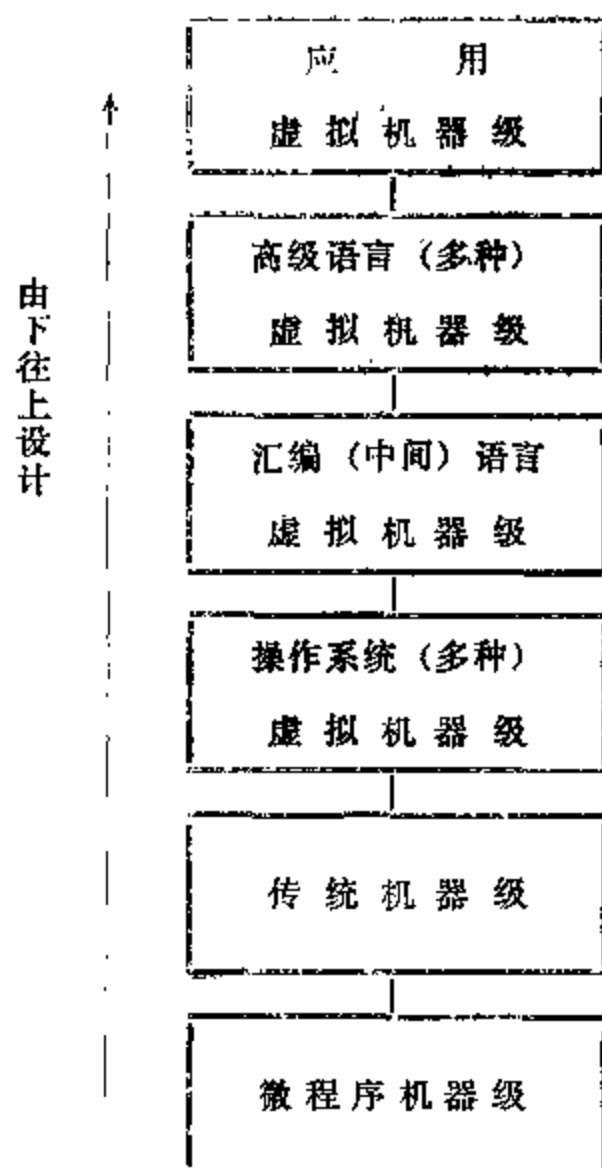


图 1.10 计算机系统由下往上的设计方法

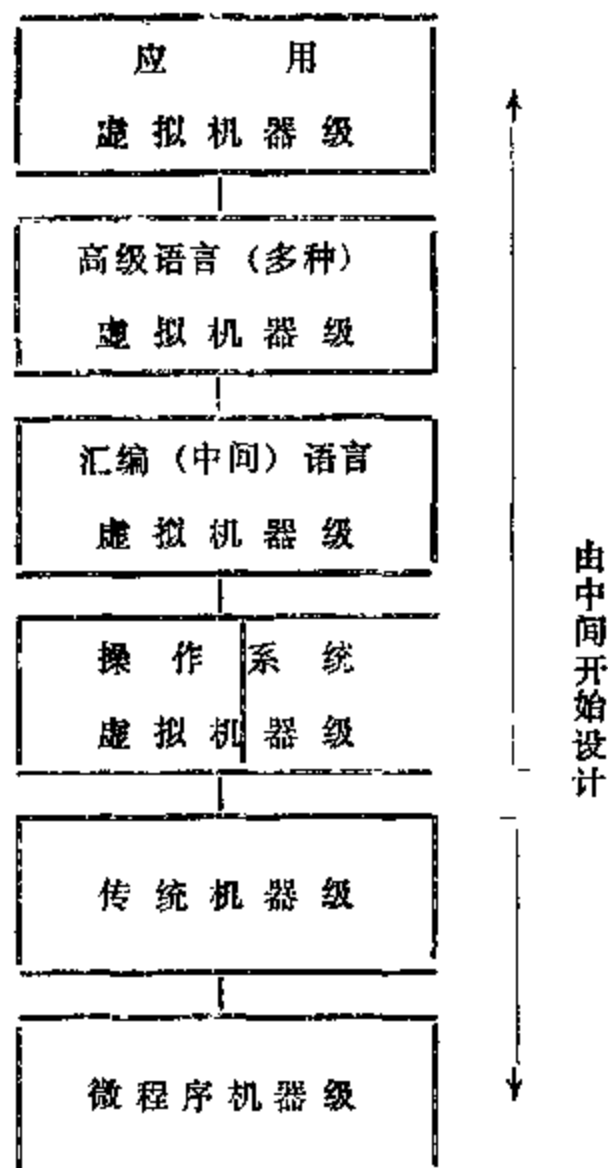


图 1.11 计算机系统由中间开始的设计方法

设计往往是“被动”的，立足于这些硬件已不能改变，而不管某些硬件的设计对于软件的实现是多么的不利，也不管本来只需少许增加某些硬件功能就能使软件设计得以简化的这种情况。其二是这样研制出来的硬件机器的某些性能指标可能是不确实的。传统机器级的“每秒运算次数”指标就是一例，虽然它是硬件设计的主要指标，但不同指令系统、不同 I/O 联接方式，其“每秒运算次数”的实际作用会差别很大；又如，若指令系统中有面向操作系统或面向语言编译的指令，则它们在缩短操作系统内务操作时间和加快高级语言程序编译过程中所带来的好处，往往会比增加“运算次数”百分之十到百分之二十还要大。而采用由下往上的设计方法，这些是不易被注意到的。

软件设计与硬件设计的相互分离与脱节是上述“由上往下”及“由下往上”设计方法的

主要缺点，而“由中间开始”的设计方法（图 1.11）是克服这个缺点的正确途径。

“中间”是指的前述多级层次结构的某个中间点，它应是软、硬的交界面。“由中间开始”的设计方法要求先定义好这个交界面，定下计算机系统的哪些功能由软件实现，哪些由硬件实现，这既要立足于能用得上的器件的品种和集成度，又要考虑到可能的各种应用的基本解题算法对机器结构的要求以及常用的数据结构等。

图 1.11 中，中间点取在传统机器级与操作系统机器级之间，是在相当一段时期以来所采用的。这个交界面是要既从硬件要求、也从软件要求来确定，即既从计算机设计者的角度，也从程序设计者的角度综合而定。之后再由中间点分别往上、往下设计。这就比较容易克服软、硬件设计相互脱节的毛病。但要注意，软件与硬件的交界面并不是只能在传统机器级与操作系统机器级之间。一方面，大规模、超大规模集成技术的迅速发展，硬件价格在不断下降；另一方面，软件设计的时间、费用增大，日益成为计算机系统发展的障碍，再加上对软件基本操作的更深入认识，这个交界面愈来愈有着上升的趋势。这和器件随集成度的提高不断上升“占领”原硬件设计领域的状况有相似之处，从下面的分析可以看清这点。

1.2-4 器件、硬件、软件功能分配的演变

现在来分析在按器件发展划分的第一代到第四代计算机系统中，从元件研制直至语言设计的整个过程中，器件、硬件、软件功能分配的演变。

图 1.12 中，第一代指的是电子管的，第二代是晶体管的，第三代是小规模集成电路（SSI）的，第四代是大规模集成电路（LSI）和部分超大规模集成电路（VLSI）的，“今后”可能是普遍采用 VLSI 的机器。“计算机组成”指的是计算机结构的实现。还有：“器”指的是器件设计及器件生产；“硬”指的是硬件设计；“软”指的是软件设计；“语”指的是语言（从第二代以后即为高级语言）设计。其它符号在下面的分析中再解释。

级号	各级的任务	按 器 件 发 展 划 代				
		第一 代	第二 代	第三 代	第四 代	今 后
9	语言的设计	硬	语	语	语	语/结合
8	语言的实现	硬	软	软/硬支	软/硬	结合
7	操作系统的设计	硬	软	软	软	结合
6	操作系统的实现	硬	软	软/硬支	软/硬	结合
5	计算机结构	硬	硬	硬/软	硬/软	结合
4	计算机组成	硬	硬	硬	硬/器	器/硬
3	逻辑设计	硬	硬	硬	器	器
2	线路设计	硬	硬	器	器	器
1	元件研制	器	器	器	器	器

图 1.12 从第一代到第四、五代各级任务分配的演变

对第一代机器，其语言即为机器语言，所以从语言的设计直至线路设计全由硬件设计者承担。当时，当然无所谓操作系统。

从第二代机器起，高级语言开始被采用，它是经编译程序软件变换成机器语言程序，再由硬件实现；由此引出了软件与硬件及它们的分工。由于高级语言的设计是面向题目或算

法，表中把“语言设计”和“软件设计”区分开。从这一代开始，批处理等操作系统也逐步形成。

到第三代机器，器件设计“占领”了原硬件设计中的线路设计领域。语言和软件有了很大的发展，就它们和硬件的关系来看，一方面出现了有助于语言实现及操作系统实现的辅助硬件(以后再叙述)；另一方面，软件概念和技术也被引用于硬件设计，即在硬件设计中采用了微程序技术(虽然微程序概念早在第一代时就已提出，但真正被应用于大量生产的机器是从第三代开始)。这二方面用“软/硬支”及“硬/软”表征：“软/硬支”表示语言和操作系统的实现虽是经软件实现，但有的机器已有了直接支持这些软件的硬件；“硬/软”表示或是采用一般的硬件设计，或是采用有软件概念的程微序设计。

进入第四代，原属硬件设计的逻辑设计中的绝大部分也被器件设计“占领”了，而且甚至属计算机组成级的中央处理机有的也已变成了单片微处理器器件，“硬/器”就是表示计算机组成虽大部分仍由多片器件搭成，但已有直接用单片微处理器实现的。这时，在语言的实现和操作系统的实现中，辅助硬件进一步发展，甚至出现了全用硬件来实现语言编译和操作系统功能的机器，“软/硬”就用于表示这点。这时计算机结构的设计开始认真地考虑软件的要求，考虑如何能简化软件的设计和维护。

至于今后呢？目前正在形成和讨论中。看来，VLSI的大量应用和发展不仅是在计算机组成中广泛采用各种处理器(如中央处理器、I/O处理器等)片子，而且还促使软件和硬件紧密结合去实现原是系统软件(如语言的编译和操作系统)的功能。这不只是为软件提供各种辅助硬件，而是从如何优化性能来分配软件和硬件的功能。从一开始就要确定哪些功能适于由硬件实现，哪些适于由软件实现，它们之间如何密切协调配合，而不是硬件设计只提供目前型式的机器语言指令系统，让软件被动地去按它设计。图中的“结合”就是表示这个意思。甚至语言的设计也可能要结合进行，图中的“语/结合”就是表示语言的设计虽主要仍由语言设计者进行，但也可能是结合设计。结合设计是今后的主要趋势。由于软、硬件的脱节与分离已成为计算机系统进一步发展的主要障碍之一，今后计算机系统结构设计者要为促进这二者的结合而努力。图中的“器/硬”表示在计算机的组成中广泛采用VLSI、LSI的功能片子(如各种处理器)，而用集成度低的片子搭成的硬件，其比例则日趋缩小。当然，如何确定片子的功能，使其既能发挥LSI、VLSI带来的好处，又能具有“通用性”，这不是容易的事(下面在§4中再来分析)。

综上所述，我们所要讲述的应是既从程序设计者角度，又从计算机设计者角度看的计算机系统结构，它是软、硬的交界面，它所要研究的是软、硬件的功能分配以及如何最佳、最合理地实现分配给硬件的功能。这就是本书所用的计算机系统结构的定义。

我们既要分析从计算机设计者角度看的计算机结构在近三十年来的进展，着重于如何最佳、最合理地实现分配给硬件的功能；又要分析从程序设计者角度看的计算机结构的进展，着重于分析“软、硬结合”进一步发展的各种可能性。器件的进展是计算机结构发展的基础和最活跃的因素之一，这点将在§4进行分析。

在转入§2之前先来 分析 大家所熟知的计算机的分型(分成巨、大、中、小、微型机等)与计算机系统结构的关系。

1.3 计算机的分型与结构的关系

本小节不是着眼于讨论计算机分型的含义和作用，而是主要从系统结构的观点来分析巨

型机、大型机、中型机、小型机和微型机的差别所在。

1.3-1 按什么分型

大家对计算机的这种分型是知道的，可是它们是依据什么来分型呢？是按照速度、按照性能、按照大小还是什么别的因素和指标呢？

是的，一般地说，巨型机的速度比大型机的快，大型机的又比中型机的快，还可依次下推。但实际情况并不如此单纯，例如从机器的主频来看，完全可能出现微型机的主频比小型机，甚至比中型机高，以致接近于大型机的情况；又如从每秒运算次数来看，那小型机的每秒运算次数完全可能比中型机的高，而却可能又比微型机的低，况且，过去（例如十五年到二十年前）大型机很费劲才能达到的主频和运算速度却是目前小型机，甚至微型机都可以达到的。因此，只按速度分型并不确切也不适宜。当然更不能按体积大小来划分，因为过去需占几个房间的机器的功能还不如现在只有一个机柜的小型机；而目前采用 VLSI 片子的大型机（如 Amdahl 470/V6），其中央处理机的体积就比同期的中型机还小。

那么，是否是按性能划分呢？在同一个时期看，性能的高低确是符合从巨型到微型的次序，这也是目前比较普遍的看法。当然，“性能评价”并不简单，它也是计算机系统结构研究中的重要课题之一。以什么来衡量性能的高低，这里只能结合计算机的分型很简单地讲讲。从硬件来看，除主频外，性能可以表现为字长（一般地，大型机的字长比小型机的长，小型机的又比微型机的长）；数据的类型（目前微型机只有定点表示，小型机已增加浮点表示，而大型机则不止有定、浮点表示，还开始有向量、数组等表示）；主存容量与编址空间（大型机的主存容量必定比小型机的大，而且其最大允许编址空间可以比主存实际容量大得多）；存贮体系（大型机有着从超高速缓冲存贮器、主存贮器、磁盘存贮器直至磁带存贮器的完整层次，而小型机则一般没有超高速缓冲存贮器）；I/O 能力，指的是 I/O 处理器的处理能力以及所能连接的 I/O 设备数量（在这点上，大型机比小型机要明显的强得多）；指令系统（大型机的指令系统比小型机、微型机的丰富）等。从软件来看，大型机比起小型机、微型机配备了更多种类的高级语言，更完善的操作系统（不只有实时，分时操作系统，还有数据库管理系统，多处理机操作系统等）和更多的用户程序包。另外，大型机在提高可靠性和可用性上采取了很多措施，如纠错编码、发现错误能重复执行、诊断技术、各种设备冗余措施等等。

但是我们却不能就此肯定地说字长超过某个值（如 32 位），或有某种数据类型，或超过某个主存容量，或是具有超高速缓冲存贮器、虚拟存贮器，就说这是大型机。也不能根据有何种指令系统，或是否有以及有多少种高级语言，或有何种类型的操作系统以及具备什么样的可靠性措施来划分机型，如此等等。因为所有这些都只是在某一段不长的时期内各型机器的表征。随着时间的迁移，尤其是随着器件技术的进展，低档机器完全可以演变成具有原高档机器的性能。近几年来，这方面的例子多得很。例如，微型机最初只有 8 位字长（甚至 4 位）、现在 16 位（目前是一般小型机的字长）的已有好几个型号，预计八十年代初 32 位（几年前是一般大、中型机的字长）的必将会出现；小型机刚成型问世时，只有定点表示，而现在就已有浮点表示了；前几年只是大型机才有的超高速缓冲存贮器和虚拟存贮器，最近好几个型号的小型机也都有了；在指令系统的类型和条数上，目前有的微型机不仅已超过几年前的小型机，而且甚至超过了当时的中型机，特别是在堆栈型指令方面。又如，最初的微

型机只有汇编语言,当时曾有人预言,由于微型机字长只有 8 位,主存容量又大不了(否则,当时的主存费用与处理器价格相比,会过分不协调),因而难于实现高级语言的编译。然而几年后的今天,不少微型机都已配上了 FORTRAN、COBOL 等高级语言。

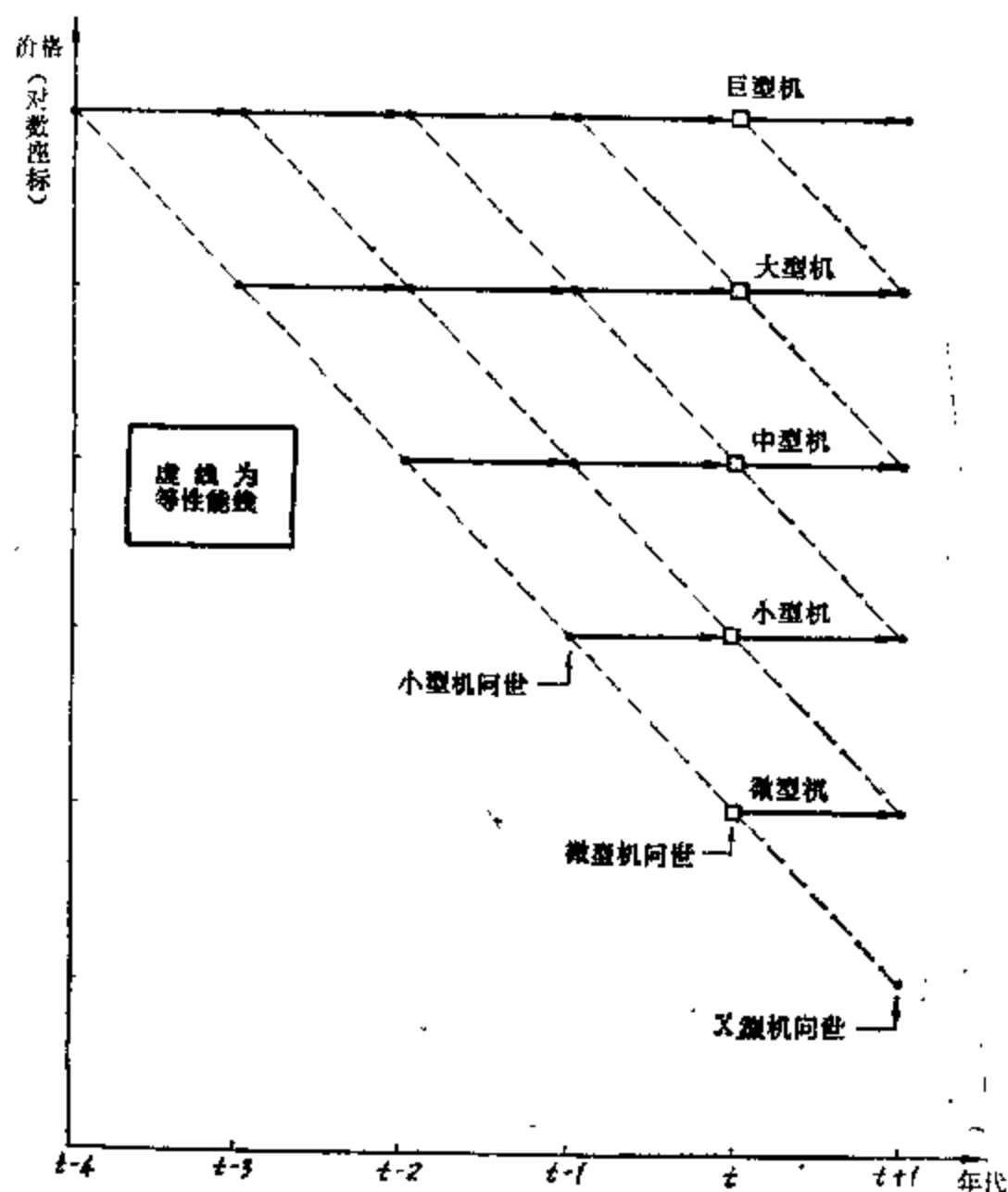


图 1.13 “分型”与价格和性能的关系示意图

综上所述,包括速度在内的“性能”虽是表征机器分型的指标,但它是动态地在不断地变化着,因此有些人就认为,能区别各型机器,并能比较长期不变的看来还是“价格”,即从长期观点看,从巨型机到微型机是以价格来划分。“以性能表征”和“以价格划分”这二者关系密切,可用图 1.13 示意。

图中的 t 表示年代,各型机的价格按对数座标近似的认为是等差的(实际情况当然远比此复杂),每型(或称每挡)分别对应于不同的性能等级(可用前述那些性能指标表征)。平行的虚线表示等性能线。此示意图可以清楚地反映各型机器的性能随时间(实质上是器件技术的进展)而演变的状况。例如,在六十年代末期(可近似对应于图中的 $t-1$)小型机(如 PDP-8 等)问世,其性能几乎与目前(对应 t)一般的微型机相近,但价格却下降了一挡;同样,以 16 位为主的小型机,在价格几乎不变的情况下,其性能正日益往中型机提高,32 位的小型机(如 VAX-11/780 等)已经出现,而五十年代末期(近似对应于 $t-3$ 到 $t-2$ 之间)要花与目前大型机几乎相同价钱买的机器,其性能却和目前的小型机相近(也许其字长要比目前小型机的长)。又如,当前微型机有二个发展趋势。一个是尽量利用 VLSI 的进展,在几乎不提高价格的前提下提高其性能,往小型机靠。另一个是维持性能大致不变,但却尽

量降低其价格，这可能出现更低挡的计算机，以更进一步扩大计算机的用途，“X型机”就是指这种机器。当然，就某一段时期来讲，每型机器所对应的性能相对稳定，具有比较明确的含义。

由此可见，计算机工业在处理性能和价格的关系上可以有二种途径。一是维持价格不变，但充分利用器件技术等进展去不断提高机器的性能，即沿图中的水平实线发展；另一是在性能基本不变的情况下，利用器件技术等进展去不断降低机器的价格，即沿图中的虚线往下发展。这二者对于计算机的发展都有意义，后者对于促进计算机的推广和大量使用，又反过来使计算机工业得以如此迅速的发展起着重要的作用。

1.3-2 系统结构与分型

上面是就各型机器的性能和价格关系进行分析的，由于机器性能和系统结构是密切相关的，所以上述分析已经联系了系统结构。下面再着重于从系统结构的观点来分析各型机器的差别。

与各型（挡）机器的性能随时间下移的情况相似，低挡机的计算机结构实质上是引用或甚至是照搬高档机器的计算机结构。例如，最初的小型机、微型机都比较简单，而后，就在器件的进展和价格允许的范围内逐步引入五十年代末期前、后问世的那些机器的系统结构特点。拿目前的微型机、小型机与五十年代末期、六十年代初期间世的机器相比，它们的系统结构几乎没有什么差别。

例如，从指令系统的操作种类来看，在算术运算、逻辑运算、访存操作，寄存器间操作、转移操作、甚至 I/O 操作等方面都几乎是一样的。又如，从指令系统的编址方式来看，直接编址、间接编址、变址编址、相对编址、基址编址等方式五十年代末期的机器就有了；从机器所具有的数据类型来看，定点数、浮点数、逻辑数、字符串等当时也有了，而浮点数表示及其运算在小型机中不久前也已开始采用；从寄存器的组成方式来看，多累加器概念、通用寄存器概念以及硬堆栈寄存器和面向堆栈的操作当时就有了；双倍字长运算、十进制运算及二——十进制间的转换等操作当时就有了；还有，各种中断方式、条件码、状态位和程序状态字等概念当时基本上也有了。

就处理 I/O、主存和中央处理机间的关系来看，当时已有的 DMA（直接存储器访问）方式，先是引用于小型机，后来又被中、高档微型机所采用。至于当时为达到 I/O 操作与中央处理机运算同时进行的 I/O 通道、I/O 处理机等概念后来也逐步引用到小型机，只是还没充分引用而已。其它的例子还可举出很多。

这种低挡机承袭高档机系统结构的情况正符合图 1.13 价格——性能曲线所示小型机（也包括微型机）的设计原则；即以尽可能低的价格，充分发挥器件技术等进展去实现已有机器的那些结构，而不是化很大力量去研究和采用新型系统结构。上面已经讲过，这个设计原则对于扩大计算机的使用范围，发展计算机工业起着重要的作用。当然，因为它们的使用对象不同，小型机也不是全部照搬原先大型机的系统结构。另外，有的计算机厂家还是企图在小型机中采用原先大型机还没有的新的系统结构，如面向高级语言、具有仿真能力的 B-1700 机器的设计等，但至今毕竟还是很少数。还有，在总线结构上小型机还是有独创的。

至于巨型机和大型机情况就不一样了，在每个时期，一般地它们若拘束于已有的系统结构，即使把器件使用到其极限状况（主要是速度和失效率），也难以达到其高速及高性能的

设计指标，也就难于在市场上推出比别的厂家更好的巨、大型机。这样，就逼得巨、大型机的设计者非得在新型计算机结构、新型计算机组成上下功夫不可。例如，在中央处理机的重迭、流水和并行处理技术；存贮系统中包括有超高速缓冲存贮器的存贮层次结构；输入/输出系统中的通道、I/O 处理机；各种从结构上提高机器可靠性的技术；多处理机技术；以及近年来从数据表示上下功夫的阵列机、向量机和便于程序编制、存贮管理的虚拟存贮器系统等都反映了这点。又由于巨、大型机的发展是沿着图 1.13 水平方向（即维持价格基本不变，充分利用新的器件等技术以提高机器的性能）的途径，甚至沿着向上斜（既充分利用新的器件等技术，又不惜提高价格以提高机器性能）的途径，所以机器的设计者就更有可能去探索和采用新的结构和组成。因此，每推出一种新的巨、大型机，一般就不只在器件、装配技术上有很大改进，而且必然在系统结构上有新的进展或突破。

综观计算机这三十年来发展，由于器件技术的突破和进展而使机器价格持续不断地下降是其一大特点。计算机系统结构也是这样，由于器件的进展，集成度的提高和价格的不断下降使巨、大型机的系统结构不断“下移”到中、小、微型机去，而且其下移速度越来越快。例如，从大型机采用超高速缓冲存贮器及虚拟存贮器后，不到五、六年，在小型机中就已开始采用；又如，巨型阵列机问世后不到七年，就已有了可接到小型机的高速阵列处理部件。因此，本书就不是局限于讨论哪种型号机器的系统结构，而是讨论系统结构的共性问题，分析从最初的 Von Neumann 机器之后那些已在大部分机器稳定采用的主要结构以及不论是大型机还是小型机中某些有发展前途的新型结构。同时从计算机系统结构是软、硬件交界面这个观点，结合软件和应用的要求以及器件的进展来分析。

§ 2 计算机系统结构发展的回顾

本节侧重回顾三十多年来从计算机设计者看的计算机系统结构的主要进展，而从程序设计者看的计算机系统结构的进展以及其推动因素则在 § 3 和下一章进行分析。由于近代电子计算机是从所谓 Von Neumann 型机器发展起来的，我们先简述其结构特点，再分析在其之后结构上有哪些主要进展。

2.1 Von Neumann 型机器的结构特点

尽管系统结构这个名词是在十五年之后才出现，Von Neumann 型机器的系统结构指的

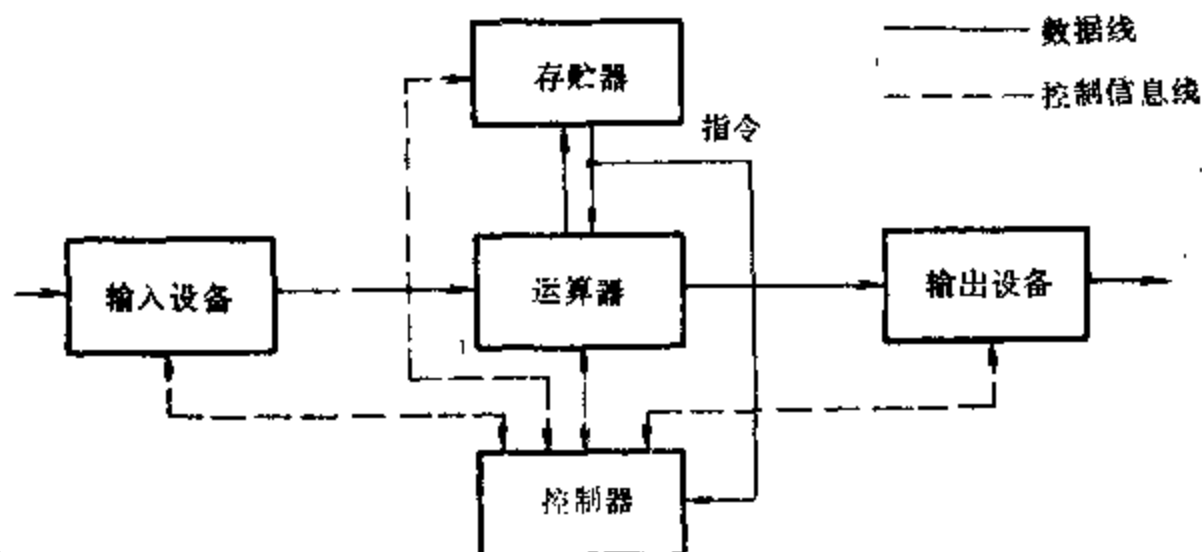


图 1.14 Von Neumann 型机器的结构

是 Von Neumann 等人于 1946 年提出来的结构。它由运算器、控制器、存贮器和输入/输出设备组成,如图 1.14 所示。

Von Neumann 型机器的主要结构特点可归结为以下几点:

(1) 存贮器是顺序线性编址的一维结构,按地址访问每个编址单元,每个单元的位数是固定的。机器的运算速度和访主存的次数关系很大。

(2) 指令由操作码和地址码组成。操作码指明本指令的操作类型,它具有的算术运算是最基本的运算,地址码指明操作数的地址。操作数的数据类型由操作码确定,操作数本身判定不了它是何种数据类型(如是定点数,还是浮点数等等)。

(3) 指令在存贮器中是按其执行顺序存贮,由指令计数器指明每条指令所在单元的地址,每执行完一条指令,指令计数器一般顺序加“1”。虽然执行顺序是可以根据运算结果改变的,但是解题算法仍然是,也只能是顺序型的。

(4) 在存贮器中指令和数据同等对待,它们本身区别不了,指令可以同数据一样送到运算器进行运算,即由指令组成的程序是可以修改的。

(5) 机器以运算器为中心,输入/输出设备与存贮器之间的数据传送都途经运算器;运算器、存贮器、输入/输出设备的操作以及它们之间的联系都由控制器集中控制。

(6) 机器以二进制编码表示,采用二进制运算。

Von Neumann 等人提出的这种结构在当时是很了不起的,它奠定了之后计算机发展的基础。虽然当时是为了解非线性微分方程而设计的,但基本上是采用这种结构的计算机却成功地应用于其它各种数值解题以及诸如信息处理系统,事务数据处理和工业控制等广泛的领域,这是 Von Neumann 等人当时所并未想到的。当然,当初设计时既没考虑到要采用高级语言,也没顾及会有操作系统以及很多应用领域的各种要求,由此引起的矛盾到三十多年后的今天就日益突出。

虽然至今绝大多数计算机仍是基于上述结构特点(从而称为 Von Neumann 型机器),但这三·多年来, Von Neumann 型机器还是有了许多的改进,下节就来简述之。

2.2 计算机系统的某些进展

本节回顾从计算机设计者角度的计算机系统结构三十年来的某些进展。

2.2-1 运算器与运算方法

运算方法指的是基本算术运算(如加、减、乘、除等)在运算器中的实现方法,或称算法。采用什么样的算法与运算器的结构密切相关,这二者的设计是相互影响的。

Von Neumann 型机器的运算器最初只有加法器和寄存器,移位操作(左、右移一位)是经加法器输入端的斜送实现。乘、除法运算是由一串相加(相减)和移位操作实现。

五十年代对运算器和运算方法的研究给予了很大的注意,取得了不少至今还在应用的成果。直至六十年代,对运算器和运算方法的研究着重于改进加法器的逻辑结构以缩短加法时间,设计快速的乘、除算法及浮点运算等。

从逻辑结构上缩短加法器的加法时间主要是把串行进位改成先行进位,即高位数的进位信号不是由低位数一位一位地进位来获得,而是由低位相加数经由逻辑组合线路直接形成。这个逻辑组合线路所需级数和数的长短有直接关系。若数的位数较长,往往把整个数分成几

组，组内采用先行进位，组间采用串行进位。当时，就组的大小和数的长短之间的合理关系，以及如何把各组再进行组合等等问题进行了不少的研究和分析。但使加法时间由1955年Pegasus机器的300微秒缩短到目前的几十毫微秒的关键是器件的巨大进展，它使每级门延迟及连线延迟极大地缩短，而在逻辑结构上基本没有多大变化。由六十年代起，从逻辑设计上来缩短二进制数加法器进位时间的研究基本上再没有什么进展了。

加快乘、除法算法的进展也主要是在五十年代取得的。对于用一串相加和移位操作实现的乘法运算，若能免除每次相加的进位动作，则会大大加快乘法运算。在乘法运算中为此设计了“保留进位加法器”，它对加快乘法运算起的作用很大，再加上二位一乘（即每次乘二位）、四位一乘、八位一乘直至十六位一乘就使得乘法速度大大加快。七十年代机器的各种快速乘法器基本上都是采用这种算法。至于除法运算，多于二位一除（即一次上两位商）其实现较难，好在整个运算中除法运算所占比例很小，在一般通用机中采用二位一除，甚至一位一除都关系不大。若要求较高的除法速度可以采用所谓“跳1跳0”算法（即一串“1”或“0”的商可以一次获得），若要求有很高的除法速度，对于有高速乘法器的运算器可以采用所谓“迭代除法”，即采用 $x/y = x \cdot (1/y)$ ， y 的倒数是用迭代方法求得，若辅之以用查表法求得前几位商及迭代用的某些系数，迭代的收敛速度可以很快。不过，用迭代除法求得的商，其精度有损失。其实，就是乘法运算，若不取双倍长乘积，而是缩短成单倍长，其精度必然也会有损失；还有，多位乘法也会因附加位的限制造成精度损失。快速乘、除法算法在速度和精度损失之间往往是会有矛盾的，设计时要注意这点。我国DJS-240、260机都是采用多位乘法和迭代除法的。有关快速乘、除法算法可见有关文献。

虽然Von Numann当初反对使用浮点表示，但因为浮点数表示的范围大得多，可以大大简化以致免除运算中比例因子的确定和修改，因此五十年代起采用浮点的机器愈来愈多。之后，标准高级语言中也普遍具有这种数码表示。浮点加、减法运算比定点的要复杂，因为需要经过对阶、尾数加或减、舍入、规格化等步骤，这里需要有多次加或减及移位操作。如何加快浮点数的运算速度在五十年代和六十年代进行了很多研究，为了缩短对阶和规格化中用于移位的时间，一是减少所需移位的次数，二是设计多位一移的移位器。由于器件价格的降低和可靠性的提高，六十年代末开始已有一些机器采用全移位器（或称鼓式移位器），它能实现全字长一移，即不论移多少位都只需移一次，花费同样的时间；它还可实现循环移位和算术移位。当然又可进一步提高浮点运算的速度。DJS-240、260机都设置了这种全移位器，它对提高速度的好处当然不仅是在浮点运算上。不过，如同单倍长定点乘法运算会有精度损失一样，浮点加、减法运算也会因对阶时把右移出运算器规定字长的那部分视为零而造成精度损失。舍入及在规定字长之外再加附加位是减少损失的办法。舍入有各种各样的方法，这里存在有舍入误差与舍入操作时间的矛盾。在设计运算器和确定浮点运算方法时，必须注意精度损失，因为它对程序设计者及计算机的使用者是“透明”的，它是由运算器内所用运算方法和舍入方法来确定的。舍入方法的改进直至目前还是浮点运算方法研究的课题，试图研究出舍入误差尽可能小、便于硬件实现和舍入时间又短的方法。有关精度损失的分析，在第二章还要讲到。

如何提高运算器可靠性以保证运算结果正确也是运算器设计中的主要课题。用编码方法去检测运算结果的正确性是一种办法。一些机器，如DJS-220、260，采用“奇偶预测”来校验运算结果，这一点在“可靠性”一章中会详述。此外，还发展了面向堆栈的运算器结

构，我们在下一章也会讲到。

2.2-2 通道与 I/O 处理机

图 1.14 的 Von Neumann 型机器，I/O 与存贮器之间的联系要通过运算器，其操作受控制器集中控制，而且是由 I/O 指令所构成的 I/O 程序执行。可是 I/O 设备的速度比运算器、控制器甚至存贮器的速度慢几个数量级，使得 I/O 在工作时，其它部分的时间利用率极低。因此很快地就把这种结构改进成在 I/O 与存贮器之间设置直接通路，而且逐步把对 I/O 和存贮器的集中控制改成分散于 I/O 和存贮器中的分布控制，目的是使运算器能与 I/O 同时重迭地运行。I/O 控制系统的功能由简单到复杂，先是把 I/O 与存贮器之间在成组传送数据时的“字数(这组共有多少个字)控制”和“存贮器地址递增(这组数据存在存贮器内一组相邻的单元)”控制功能接过去，即把判字数与修改存贮器地址的软件硬化成硬件实现，并把它从中央控制器移到 I/O 控制器，此即所谓 DMA (直接存贮器访问) 控制。在进行 DMA 传送时，运算器可以并行的进行运算(这里，当然会有访存冲突问题)，DMA 传送结束时，通过中断系统告知主控制器。后来，I/O 控制系统又把诸如选设备、切换、启动、终止以至数码校验等功能也接过去，并进而分解成 I/O 处理机(通道)与 I/O 控制器如图 1.15 所示。其中，中央处理机包括了前述的中央控制器与运算器。对比图 1.14 与图 1.15，机器的系统结构由以运算器为中心转变为以主存贮器(通过主存控制器)为中心。主存控制器对各种访主存申请(如取指令申请、取操作数申请、送运算结果申请、通道申请等等)进行排队和切换。

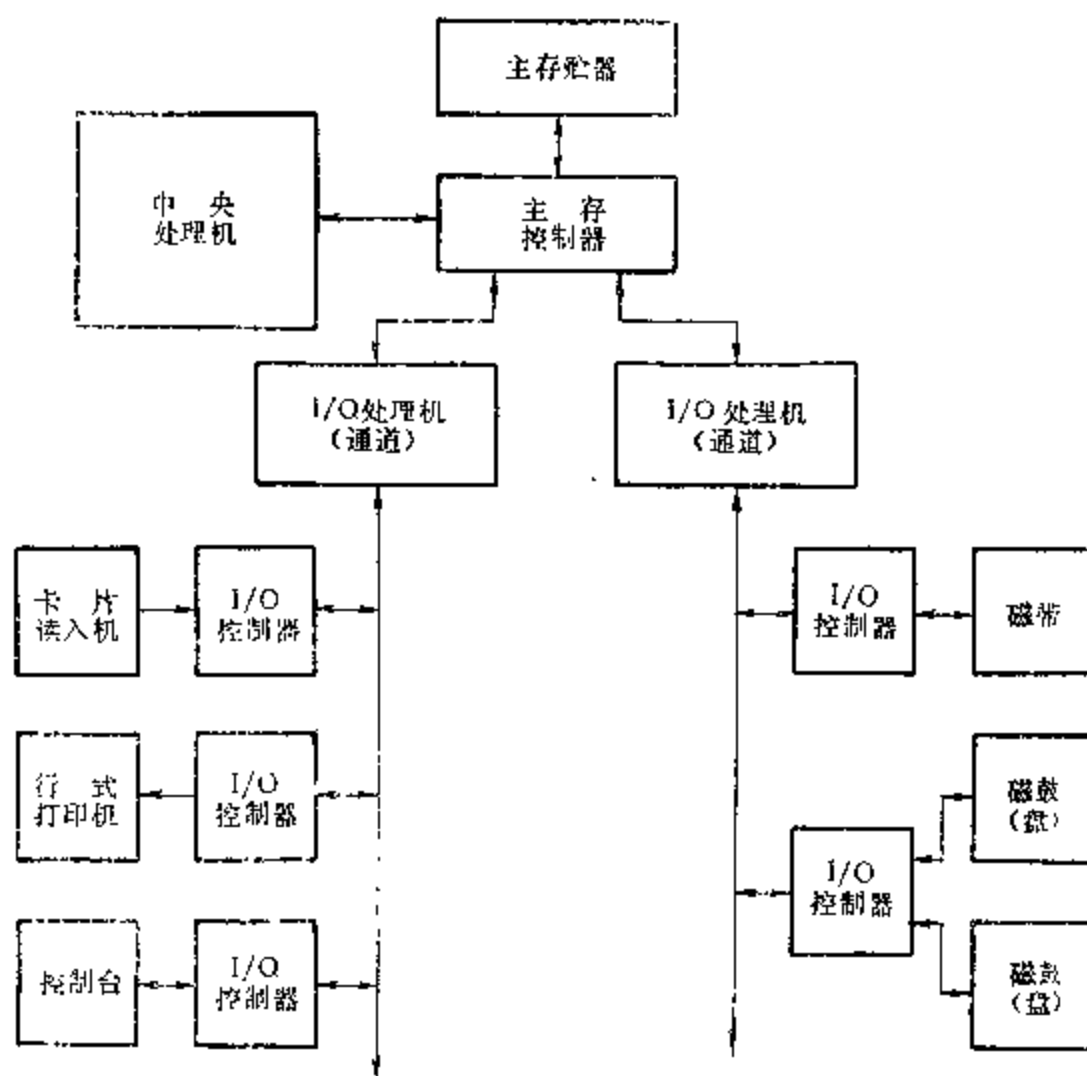


图 1.15 具有 I/O 处理机 (通道) 的机器结构

五十年代中期起，中断系统形成并逐步完善，至今它还是解决机器各部分之间及多机之间联系、同步等的主要工具。在图 1.15 的结构中，I/O 处理机就是通过中断系统与中央处理

机通讯,并由 I/O 处理机先对各个 I/O 控制器发出的中断申请进行按排。

图 1.15 的系统结构至今几乎大、中、小型机都仍在采用,只不过小型机的可能简单些而已(如其通道的功能比上述 I/O 处理机的要简单等)。这种系统结构是五十年代中、末期的机器(如 LARC, 709 等)都已具备了,而它在六十、七十年代里进展却不大。

然而这种结构的出现不论在硬件和软件上都提出了新的思路。在硬件上,这种结构把原是中央控制器集中控制的许多功能“分布”到主存控制器、I/O 处理机和 I/O 控制器去。而且 I/O 处理机内也有软件,它的控制功能是通过程序的执行来实现,I/O 处理机当然也有指令系统,只不过它不是面向运算,而是面向控制,从而比中央处理机的要简单。有的 I/O 处理机,其程序存在主存贮器,有的则在 I/O 处理机内设置有存放程序的存贮器。当然,在 I/O 处理机内必须有存放 I/O 数据的缓冲存贮器。中央处理机的一条 I/O 指令启动 I/O 处理机内的一段程序,就是说是由 I/O 处理机的程序解释中央处理机的 I/O 指令。中央处理机这条 I/O 指令通过 I/O 处理机的软件、硬件和 I/O 控制器硬件以软、硬结合的方式来实现。I/O 处理机的软件必然和硬件紧密相关,对于图 1.15 的结构,它当然是系统程序员看得见的,而对应用程序员却是透明的。

这种系统结构与图 1.14 的结构不同,只要中央处理机把该条 I/O 指令的控制信息传送给 I/O 系统后,就可继续执行它自己的运算,使得中央处理机的运算和对 I/O 系统的操作及其运行能同时进行,不必互相等待。这就为多道程序、分时处理等操作系统软件的建立提供了基础。这种软件和系统结构的紧密结合使计算机系统的各个资源尽可能不间断的运行以获得充分的利用,比较有效地解决了 I/O 设备的低速与电子器件的高速之间的巨大矛盾。当然,这个矛盾至今仍然存在,仍是推动系统结构发展的主要因素。

在研究和发展图 1.15 的系统结构中,总线结构有了发展。例如,主存控制器的主要功能就是对主存总线的分配和管理;而通道的输入输出也是经由总线,各种 I/O 控制器只要具备总线要求的统一接口都可接到总线上,这样,在机器设计好甚至运行之后仍可方便、灵活地增减所配的 I/O 设备量和类型。在六十年代,不少小型机就引用了上述总线结构,并进而发展成如 PDP-11 所用的单总线系统。总线结构至今仍然是系统结构研究的课题。

有关总线结构以及 I/O 系统在“输入/输出系统”一章还要详细讲述。

图 1.15 把控制功能“分布”开的这种系统结构实质上就是近年内所谓“分布处理”系统的原始雏型。随着 LSI 和 VLSI 的迅速发展,硬件价格的不断下降,就有可能在计算机系统内设置更多的专用于执行各种任务的处理机。有关“分布处理”的基本概念,我们在第三章还要讲述。

2.2-3 微程序

微程序的原理和基本概念已在“计算机原理”课中讲过。可以说,前述运算器和运算方法以及 I/O 处理机等方面的改进主要是围绕如何提高机器的速度;但微程序技术的出现和采用最初却是为了使控制器规整,以便于设计和生产,那怕因此使机器的速度会有所下降。

它之所以能够使控制器(参看图 1.16)规整化是由于把存贮器技术引入到控制器内(有人因此称之为存贮逻辑),从而把硬联控制器网络的复杂逻辑联结转变为控制存贮器的复杂码点(各种微指令码点),可是不论所存码点是复杂的或单一的,存贮器(不论是 ROM 型或 RAM 型)逻辑结构和工艺结构的固有规整性都不受影响。这就完全可以用大量生产的通

用ROM、RAM片子去代替原先复杂的硬联控制器网络。微程序控制器内的其它部分，如微

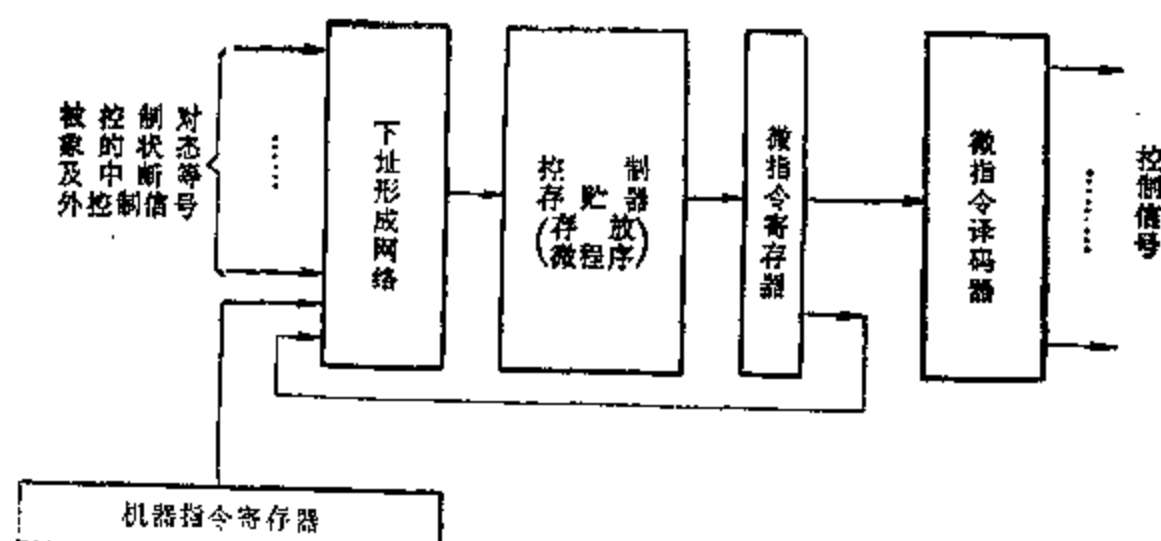


图 1.16 微程序控制器

指令寄存器和微指令译码器都是规整的；下址形成网络的规整性要差些，但其不规整性比硬联控制器网络的要好得多。规整性给控制器的设计、生产和成本的降低带来极大好处，尤其是适合于大规模集成电路发展的需要。大家知道，复杂的硬联控制器网络，就是使它大规模集成化也只能应用于一种机器；可是，对于微程序控制器，器件厂就可能生产出大规模集成电路这样的微程序控制器片子，只要配以不同的控制存储器内容（即不同的微程序）就能够用于解释好几种机器的指令系统，因此提高了所生产片子的通用性，位片式微处理器就是一例。结构的规整性还带来了便于诊断和维护的好处。微程序的具体设计和校验与微程序控制器其它部分的制造可以在时间上重迭进行，以缩短机器的研制周期；而对硬联控制器，必须等全部设计完成后才能生产。采用微程序控制还可带来便于调机的好处，在调机中发现错误，往往不必更改连线等硬结构，只需修改微程序，改变控制存储器的内容就可以纠正，这是硬联控制器无法做到的。

但是，随着微程序技术的应用和发展，其意义已经超出了规整化的目的。微程序控制对系统结构发展的影响并不主要是由于它的规整化，而是由于它把软件的概念和技术引用到了机器指令级，引用到原是全硬实现的那些环节，以实现硬件的软化，这给机器的系统结构带来了灵活性。大家知道，硬联控制的机器一旦制成之后，其机器指令系统也就定死了。然而微

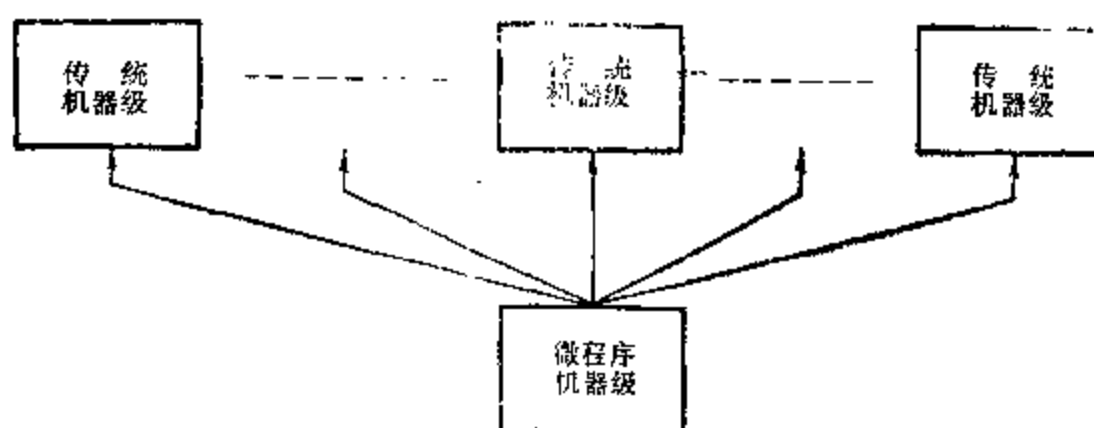


图 1.17 微程序可提供可变的多种机器指令系统

程序控制的机器，由于每条机器指令是由一段微程序解释，而各种机器指令总是可以由一串基本微操作（如各寄存器间的传送；经加法器、移位器、乘法器的传送；总线控制；访存操作等等）实现。因此，通过增加或修改微程序就可提供可变的机器指令系统，以至于可能在一台机器上提供多种型号机器的指令系统，如图 1.17 所示。

这种机器指令系统可变的灵活性给系统结构带来了好处。例如，可在机器指令系统内增加各种复合指令和宏指令（如多倍长运算、十进制运算、字符行处理、矩阵相乘、求多项式、开方、三角函数运算、查表、字节测试等等），它们的执行速度比用机器子程序实现要快得多（几倍甚至10至20倍）。

对于微程序控制在速度上的得失，需要仔细全面分析。一方面，由于对应机器指令的每一拍，微程序控制都需访问控制存储器，取得微指令后才能发出控制信号；而足够容量的控制存储器，它的读数时间又不能很快，会占到拍宽的相当部分，因此微程序控制比起硬联控制在速度上会有损失，正是因为这个原因，超高速机器一般难以采用微程序控制。另一方面，对许多基本运算，如上所述，固件实现要比软件实现快。

因此，系统结构设计者要着眼于如何发挥微程序之所长。看来，重要的一环是如何使微程序从只能由机器设计者编制和装入控存（即其微程序对机器使用者及程序设计者是透明的）进展为机器用户也能根据他的需要编制和装入，即具有所谓用户微程序的功能。一般称具有用户微程序能力的机器为微程序可编机器，而微程序只能由机器设计者编制的机器称为微程序机器。只有提供微程序可编机器，用户才能在固件实现和软件实现之间进行比较和权衡。当然，真正的微程序可编机器不只是要有可写入的控制存储器，更重要的是要能为用户提供完善的微程序软件，至少要有微程序汇编语言以及模拟检查程序和装入程序等软件，进一步就要求有微程序高级语言。一定要认识到，就是一般的微程序机器，机器设计者也不应靠效率低、错误多的手工编制，因此同样需要有微程序软件。

我们在这里主要是围绕系统结构的实现来分析微程序的，就是引出程序可编机器也只是从速度这一点来讲。然而微程序技术的作用以及它对系统结构可能的影响远不止这些。我们以后还要讲到。

从计算机设计者角度看的计算机系统结构三十年来的主要进展还有许多，我们就不在这里细述了。其中，在提高速度方面起重要作用的有“重迭与流水”和“存贮层次”，我们将分别在第三、五章讲述；另外，计算机可靠性的提高是推动计算机发展的主要因素，从系统结构着手来提高可靠性也取得了很大进展，我们将在第六章讲述。还有，原来的Von Neumann型机器是单处理机系统，随着提高速度和可靠性等的需要，多处理机和多机系统自七十年代之后有了蓬勃的发展，我们将在第三章开始分析，并在第七章集中、详细地讲述。

我们在前面已经讲过，对计算机系统结构的全面认识还应包括从程序设计者看的计算机系统结构。就是说，不能只从硬件本身的要求，还应从软件和应用的要求来分析系统结构的发展。下一节起我们就开始按此思想来讲述。

§ 3 软件与应用对系统结构的影响

系统结构设计者除了从硬件角度，从提高机器指令执行速度来设计系统结构外，还应根据软件和应用的需要来设计。我们曾讲过，系统结构设计的一个主要任务是处理好软、硬件之间的功能分配。本来，系统结构的发展应更多地适应软件与应用的需要，可是，目前的系统结构还没有很好地反映出这点。我们相信，八十年代将会更多地按照软件与应用的需要来研究和设计系统结构的。

我们已经讲过,从功能上讲,软件和硬件本来是等效的。从原理上讲,软件的功能可以用硬件来实现,硬件的功能也可以用软件来实现。早期,计算机系统的设计,由于当时的器件状况,不得不从尽量简化硬件结构出发,把更多的功能交给软件去实现。那时,器件既贵又不可靠,相对来讲,软件价格要比硬件便宜,因此这种设计原则当然是正确的。这样,作为反映硬件结构功能的机器指令系统就比较简单、基本;而应用上的更多要求则由软件去解决。长期以来一直遵循着这个设计原则,这就使得软件日益庞大。因而,随着计算机系统的发展和应用范围的扩大,积累了大量的应用软件与系统软件。由于软件的设计是编写虽易、但排除错误难、生产效率低。因此,除非重新设计能带来很大的好处,否则程序设计者是不愿意也不应当按新的系统结构、新的指令系统重新设计其应用软件和系统软件的。而没有完善软件的计算机系统是无法正常工作的。所以,近年来有更多的人认为,庞大的、不易改变的软件已逐渐成为妨碍计算机系统进一步迅速发展的“保守”阻力。尤其是三十年来,软、硬件本身的状况都已发生了很大的变化;硬件价格持续下降,可靠性日益增高,而软件价格却一直在上升,可靠性的提高却很慢。这样,从系统结构的角度研究软、硬件功能的重新分配已越来越为人们所重视。看来,以软件的复杂化为代价来达到硬件结构的简单化这个三十年前的设计原则是一定要改变的。这样我们才能更充分地利用和发挥迅速发展的大规模、超大规模集成电路等技术的作用,让新的器件技术上的成就更迅速地应用到计算机系统中去。而且,不只是用来改进计算机系统结构的实现方法,还用它来促进新型系统结构的研究和产生。

从六十年代起,如何解决程序的可移植性 (Portability) 问题是计算机系统设计者和应用者非常关心的一个方面。所谓程序的可移植性指的是一个程序可以不经修改地由一台机器搬到另一台机器上,即同一个程序可以应用于不同的环境。如果程序具有可移植性或是计算机系统设计成能实现程序的可移植性,则已证实是可靠的软件就能长期应用,不必随着机器的更新(如设置速度更高的机器)重新编制;而且按这台机器编制出来的应用软件也就能应用于其它机器。这就能大大减少编制软件的工作量(这种工作量是很惊人的,往往需几百、几千个人年),尤其是可大大节省各个计算机系统为编制功能相近或甚至相同的软件所耗费的重复工作量。同时,新的硬件技术也就能很快被采用,使新的计算机系统能迅速发挥作用而不必因为需重新编制软件化费很多的时间和精力。下面从系统结构的观点来分析这个问题。

3.1 程序的可移植性要求对结构的影响

这里讲的程序包括应用程序与系统程序。大家知道,程序可用高级语言、汇编语言及机器语言来编制。如果所有程序都是用一种统一的高级语言来编制,那程序的可移植性问题也就解决了,但问题并不那么简单,下面来分析。

3.1-1 采用统一的高级语言

如果真能统一出一种能满足各方面需要的高级语言,那至少应用程序的可移植性问题就解决了,如果操作系统的全部或一部分可用高级语言来编制,那系统程序中的这部分也可以移植。这样,各机器间的应用程序以至系统程序就可相互通用,而且,我们可以充分利用器件技术和硬件技术的进展,迅速地推出各种新型机器。

问题是至今虽然已有上百种高级语言在使用，但没有一种高级语言能够满足各方面的需要，即它们并不具有真正的通用性；甚至就在军用机这一方面，据美国有关研究报告指出，在现有语言中也还没有一种高级语言能满足大多数用途的需要，更不用说全部需要了。这是因为不同的用途往往要求有不同的语法结构和语义结构，长期以来用的比较广泛的 FORTRAN 与 COBOL 语言就是分别适用于科学计算和事务处理。而且，有的程序设计者还宁愿采用适用于他的题目，而且是他所熟悉的语言来编制其程序，那怕这种语言不是常用的。因为这可以使他的程序简练、清楚，便于检查。

统一不起来的另一个主要原因还在于我们对语言的基本结构还没有透彻的和统一的认识。对 GO TO 语句的非难和认识上的不一致就是一例，有人认为它目前的这种用法是造成程序的复杂化、不易读、不易检验和不易排错的主要原因。近年来，甚至对某些赋值语句也有这种非难。因此，要能够设计出真正能满足各方面需要的、又有相当大的发展前途、的确是比目前已有语言优越得多的统一语言（否则，人们是不会抛弃他已习惯使用的语言以及长期积累的、用原有语言编制并已被证实是正确的程序），还需要进行大量的研究工作。

目前的高级语言状况是甚至用同一种高级语言（如 FORTRAN）编制的程序，也很难在各个不同厂家的机器上通用。这是因为目前的高级语言还没有完整的、全面的统一规定，不同的机器字长、不同的“机器零”定义、不同的已置入（已定义）好功能的子程序、不同的编址空间、不同的操作系统结构都会导致用同一种高级语言编制的程序不能在各个机器间移植。而且各个厂家甚至连已有的统一规定也并不全遵守，各家往往有其自己的所谓“特色”与“方言”。如果再考虑到，在主要用高级语言编制的程序中，为了节省存贮空间或提高执行速度，有时需在部分环节采用汇编语言或机器语言（这在实时系统中并不少见，因为用汇编语言或机器语言编制的程序，其执行时间可以准确预估），那这样的程序就更难于移植了。

这些都给高级语言程序的移植带来了麻烦。因此，目前每种机器都得配上对应于多种较为通用的高级语言的编译系统。对同一种高级语言，各个机器的编译系统又各不相同，根本不能通用。这些，当然不利于系统结构的发展。

上面我们讲了统一高级语言的困难，然而这仍然是一个方向。标准的、单一的（甚至二、三种也好）、统一的高级语言对于大大节约软件的研制人力、物力和费用，对于加快人员的培养等等都有着很重要的作用。

至于在不同型号机器之间采用相同的汇编语言，这对于解决用汇编语言编制的系统程序的可移植性是有好处的；然而，由于汇编语言和系统结构是如此密切相关，企图通过配上不同的汇编程序使得不同型号机器具有相同的汇编语言，这只有在结构相同或相似的机器之间才可以做到；对于结构差别大的，这是几乎做不到的。因此，通过统一汇编语言，只能做到程序在具有相同结构或相似结构机器间相互通用。

此外，通过统一机器语言来达到程序的可移植性，这和统一汇编语言一样，只能在结构相同的机器之间做到程序可以相互通用，这就是所谓的系列机概念。

3.1-2 系列机概念

程序设计者希望所编的程序能长期地应用而稳定不变，但器件技术的迅速发展却推动着计算机工业能很快地（如每隔二、三年甚至更短）提供新的、性能不断提高的机器。如果要

求程序设计者也如此快地按新的机器修改或重编其程序，那会使软件无法发展，显然是不行的。因为，只有软件的环境长期稳定，应用软件与系统软件设计者才能不断提高软件质量，使其日益完善。而我们刚分析过，企图通过统一的高级语言来达到程序的可移植性又走不通。这样，在1964年IBM公司就提出了系列机概念并推出了IBM360系列机。系列机概念成功地在—个公司范围内解决了软件要求环境稳定与器件、硬件技术迅速发展之间的矛盾。系列机概念指的是先设计好—种系统结构，而后软件设计者就按这种系统结构设计它的软件，硬件设计者就按器件状况和硬件技术研究这种结构的各种实现方法，并按照速度、价格等的不同要求，分别提供不同速度的各挡机器。实际上，“系统结构”这个名词的概念就是由此而来的。在§1我们讲过，系统结构这个名词是Amdahl等人在1964年提出的，Amdahl当时在IBM公司工作，他就是按系列机的概念提出他的系统结构的定义，我们不妨重复于此：“在这里，系统结构这个术语是用于描述由程序设计者所看到的系统的属性，即它的概念性结构与功能特性以区别于数据流与控制的组成、逻辑设计及物理实现”。

那么，什么是程序设计者所看到的系统的属性呢？前面已经讲过，粗略的说是机器的指令系统和I/O的联结、使用方式，然而这种说法并不能概括全面，例如，中断系统中的很多功能就是程序员所必需看到的，而在指令系统与I/O联结方式中并没有明确地被概括进去。我们也难于简要地列出程序设计者所看到的系统的属性还应包括哪些内容。总而言之，这个属

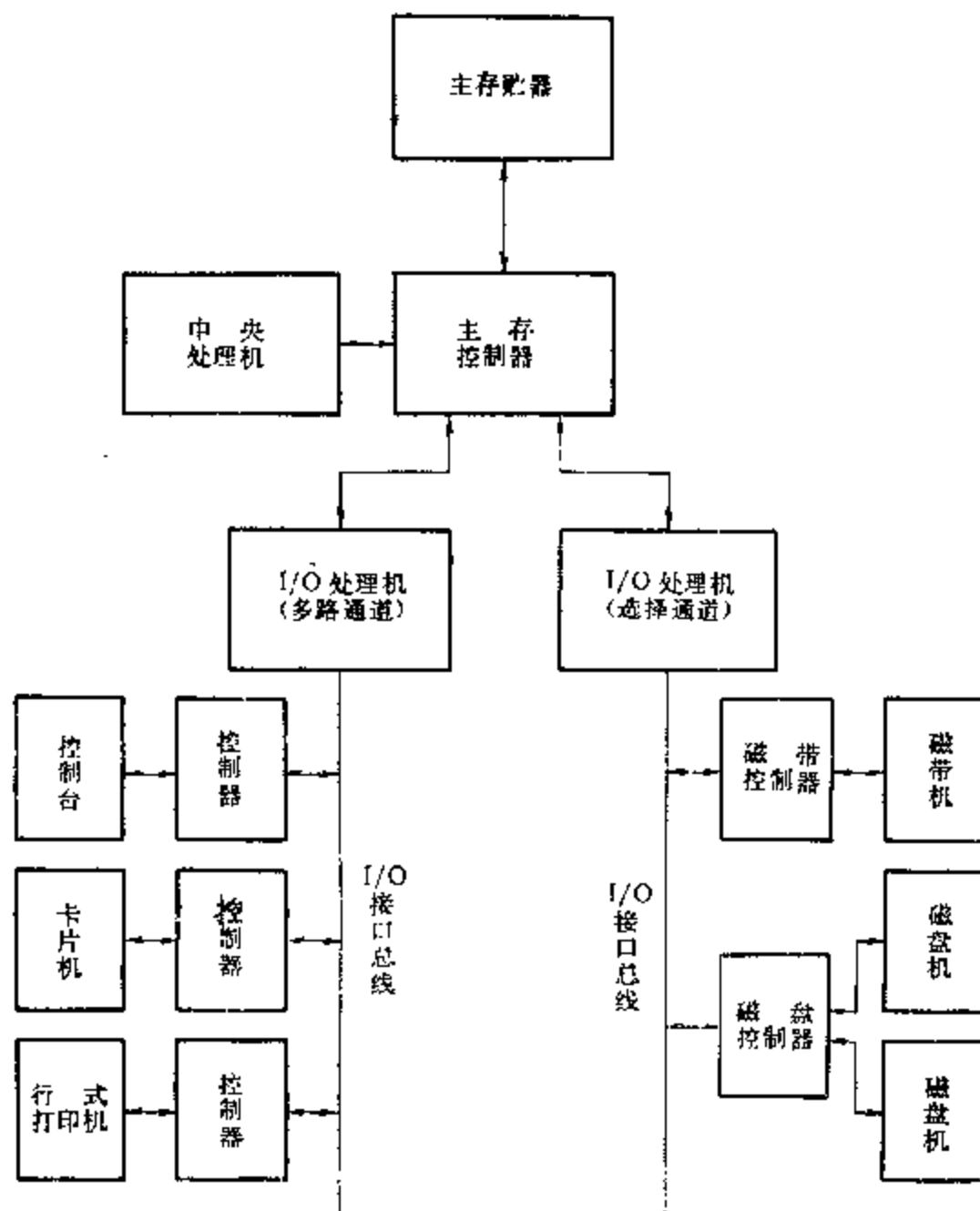


图 1.18 IBM360/370 系列机的概念结构

性的范围应是程序设计者能根据它们来编制机器语言程序和编译程序等等，而且，这些程序能够在机器上正确地运行。属性中最主要的是指令系统、数据表示及概念结构（图 1.18 为 IBM360/370 的概念结构）。这个属性实际上就是在 § 1 讲过的从程序设计者看的系统结构。在讲述系列机时，我们简称为系列结构。

在确定这种系列结构时，要非常慎重，因为这要影响到所定系统结构能有多长的使用寿命以及按此结构编制的程序能有多大的稳定性。能使用多长时间，也就是这个系列机能有多长的寿命。作为通用系列机，它应适应多方面的需要，IBM360 当时就是考虑按照科学计算、事务处理与实时控制这三方面的需要来确定其系列结构的。这样，它不只是有完整的（从当时角度）科学计算用算术运算指令，而且还有用于事务处理的字符行运算与十进制运算指令以及实时控制所需的比较完善的中断系统及某些有关的指令。在确定系列结构时要很仔细地考虑软件编制的需要，除了要考虑到便于编制系统软件和应用软件外，还要考虑发展的需要。要求在十年，甚至更长时期内，软件设计者仍然认为这个系统结构是他编制程序的好环境。

系统结构设计者在研究这个从程序设计者看的系列结构时，要考虑到它既能用简单、便宜的组成方法（如取指令与取操作数之间顺序执行的通常方法）实现；而且当采用复杂、昂贵的实现方法（如流水线方法）时，这种结构仍然能够充分发挥该实现方法所应带来的好处。一般地说，系列结构复杂一点，适于采用多种实现方法的可能性就愈大，愈不会妨碍使用新的技术。这里所讲的不同实现方法就是 § 2 讲的从计算机设计者角度看的不同的系统结构，在讲述系列机时，这种系统结构我们简称为机器结构。

系统结构设计者在实现从程序设计者看的系列结构时，一定要做到不论采用什么样的机器结构，从程序设计者所看到的机器属性不能有不同。例如，360 系列中的各挡机器，由于所要求的速度与价格不同，其数据通路（数据总线）宽度分别为 8 位、16 位、32 位甚至 46 位，如图 1.19(a) 所示；然而对程序设计者来讲，他所看到的字长都是 32 位，他所看到的定点数都是统一的半字长或全字长，而浮点数也是统一的单、双、四字长，如图 1.19(b) 所示。

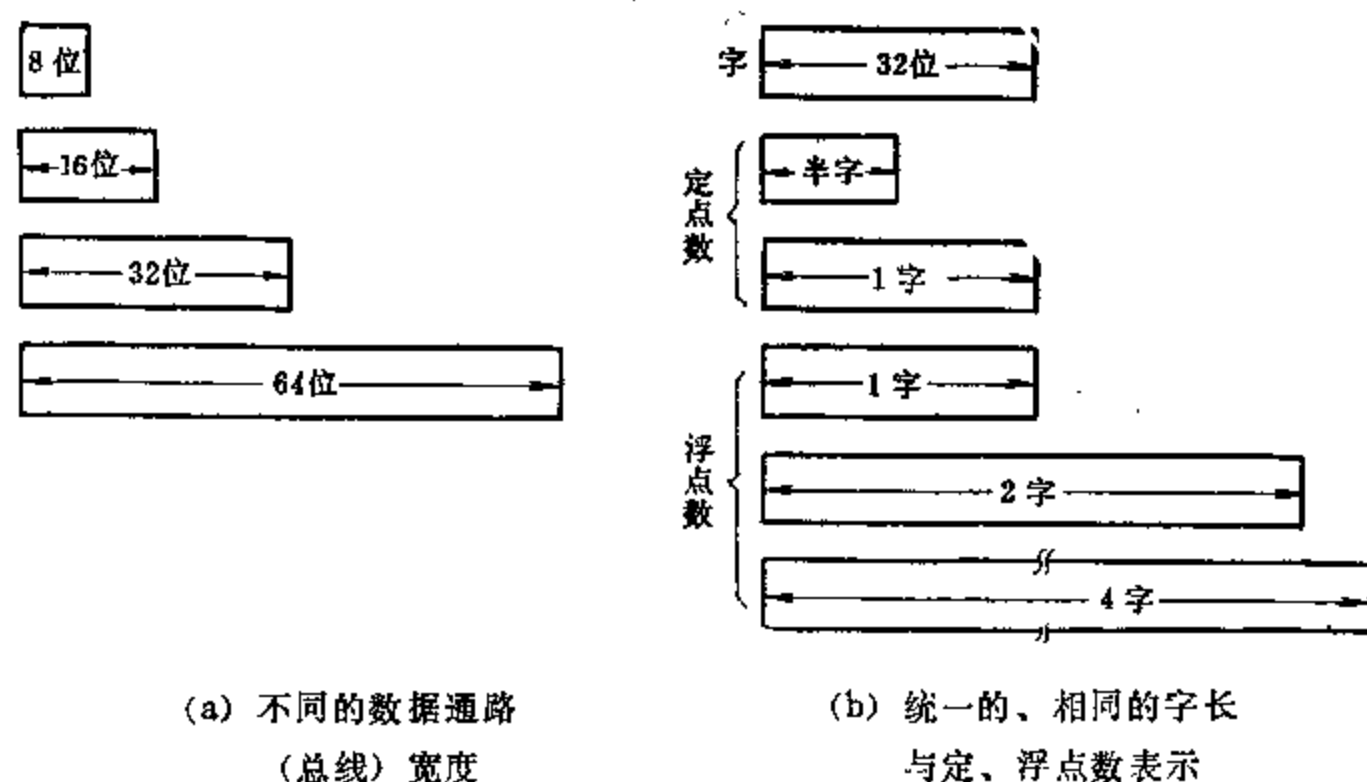


图 1.19 360/370 各挡机器有不同的数据通路宽度，但系列结构具有统一的字长与数据表示

正是因为 360/370 内各挡机器从程序设计者看都具有相同的机器属性，因此按这个属性

(系列结构) 编制的机器语言程序以及编译程序等都能通用于各挡机器。我们称这种情况下的各挡机器是软件兼容的。软件兼容指的是同一个软件可以不加修改地运行于系列结构相同的各个机器, 而且它们所获得的结果都一样, 差别只在于不同的运行时间。可以说, 软件兼容是通过采用相同的系列结构以实现程序的移植。一个系列原则上讲可以共用一套编译程序, 它能运行于从低速到高速的各挡机器, 若用 § 1 所述的层次结构来描述系列机, 则 370 系列从高级语言机器级直至实现级 (对于采用微程序控制的, 则为微程序机器级) 的对应关系如图 1.20 所示。

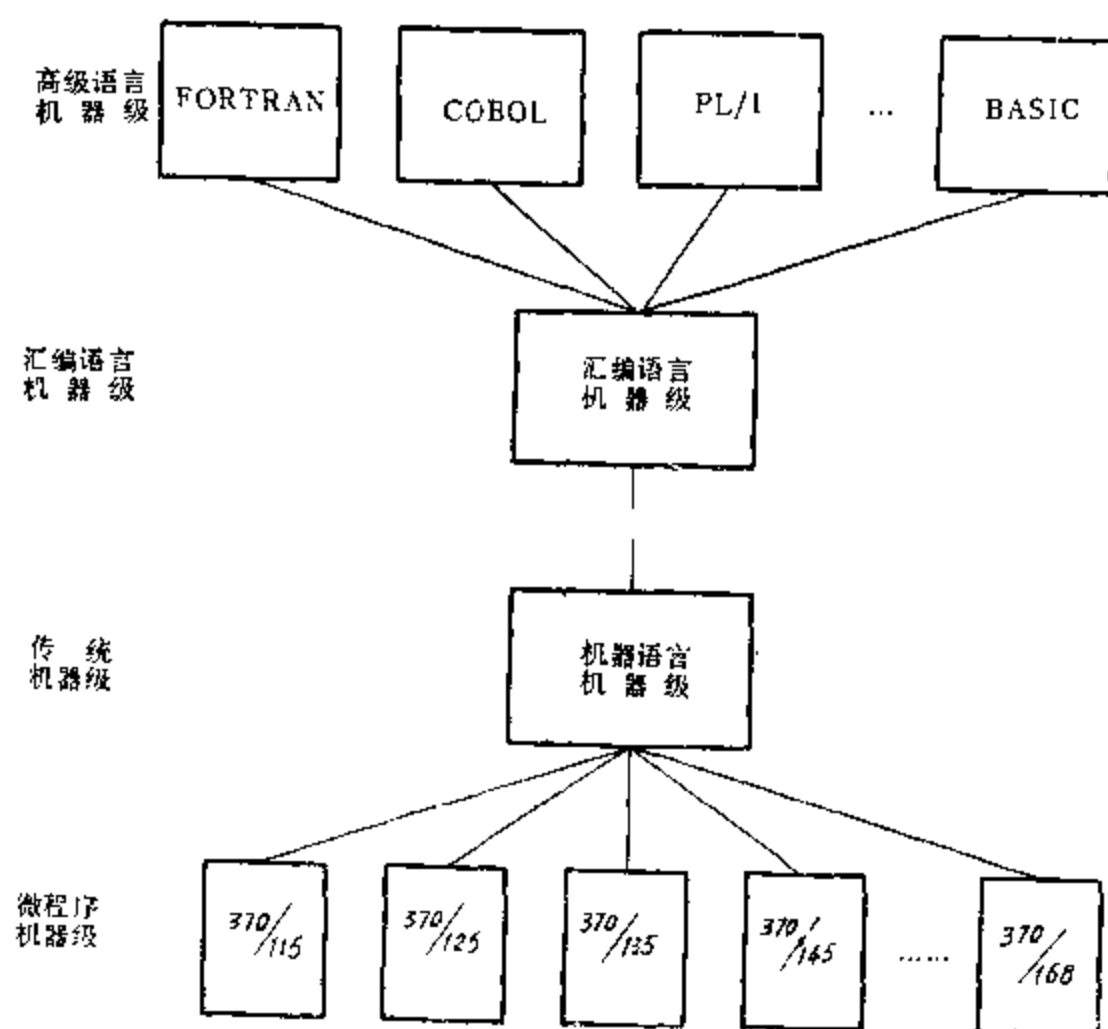


图 1.20 370 系列具有统一的机器语言级, 可以只用一套编译程序

370/115, 125, 135, 145, 158, 168 等是 370 系列内从低速到高速的各挡机器, 它们都是软件兼容的。一般来说, 一个程序 (不论是高级语言程序还是汇编语言、机器语言程序) 是能够通用于这些机器之间的。但是, 这得有一定的条件。实际上软件兼容有向上兼容和向下兼容二个含义: 向上兼容指的是按某挡机器编制的程序, 不加修改就能运行于比它高档 (速) 的机器; 向下兼容指的是按某挡机器编制的程序, 不加修改就能运行于比它低挡 (速) 的机器。系列机内的软件兼容一般是可以做到向上兼容, 但向下兼容则要看是什么样的程序, 不是都能做到的。例如, 按低挡 370/115 编制的程序肯定可以运行于高档 370/168; 然而按 370/168 编制的程序却并不一定都是不加任何修改就能运行于 370/115。此外, 系列机的软件兼容还有向前兼容和向后兼容的不同含义, 这里的前、后指的是各挡机器研制成功, 投入市场年月的前和后。向前兼容指的是按某个时期投入市场的该型号机器编制的程序, 不加修改就能运行于在它之前投入市场的机器; 向后兼容指的是按某个时期投入市场的该型号机器编制的程序不加修改就能运行于在它之后投入市场的机器。系列机是一定要能做到能向

后兼容，然而向前兼容就不一定了。

我们结合另一个成功的系列机 PDP-11 系列来说明这点。大家知道，PDP-11 系列机中的第一种 PDP-11/20 是 1970 年投入市场，按它编制的程序完全可以不加修改地运行于在它之后投入市场的 PDP-11 其它各挡，如 75 年的 PDP-11/70，PDP-11/04 等以至 79 年投入市场的 PDP-11/44，这是向后兼容。然而，PDP-11/70 比之 PDP-11/20 在指令系统上多了浮点运算指令，而 PDP-11/44 又比 PDP-11/70 多了事务处理指令。这样，若按 PDP-11/70 编制的机器语言程序中用了浮点运算指令，那当然就不能运行于 PDP-11/20，这就是，不能向前兼容。同样，若按 PDP-11/44 编制的机器语言程序中用了事务处理指令，那也不能运行于 PDP-11/70，虽然按速度分挡来讲，PDP-11/70 比 PDP-11/44 高。也就是说，就 PDP-11/70 与 PDP-11/44 之间来讲，不一定实现得了向上兼容（PDP-11/70 比 PDP-11/44 高档），但向后兼容（PDP-11/44 比 PDP-11/70 后出）却肯定实现得了。

综上所述，软件兼容本是系列机设计的基本要求，从现有系列机来看，对大多数软件，尤其是对绝大多数的应用软件来讲这点是能够实现的。但是，在有些情况下，对某些程序，向下兼容或向前兼容却实现不了。因此，确切地说，系列机软件兼容指的是要做到向上兼容与向后兼容，其中向上兼容在某种特殊情况下可能做不到，但向后兼容却是肯定要做到的。可以说，向后兼容是软件兼容的根本特征，也是系列机的根本特征。一个系列机系列结构设计得好坏，是否有生命力，就看是否能在保证向后兼容的前提下，不断改进其机器结构。

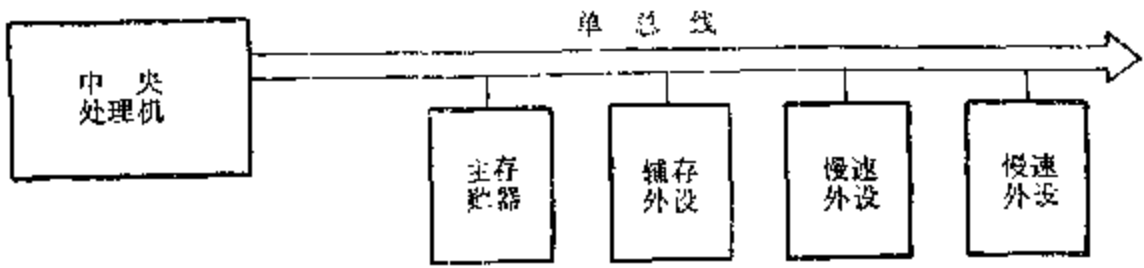
还可以看出，就是从程序设计者看的系列结构来说也还是可能随着器件价格的降低以及软件和应用的需要而扩展的。如前述的增加浮点运算指令以及事务处理指令等等。但是，绝不能任意地更改，如缩小指令系统，改变 I/O 的使用方式以至于破坏软件兼容，尤其是向后兼容。而且这种扩展应该是从提高机器的总性能出发，而不是盲目地只从提高硬件性能出发作些“灵巧性”的修改。这种扩展往往是从改进系统软件的性能出发，例如前述的增加浮点运算指令是为了提高 FORTRAN 编译程序的编译效率及其目的程序的执行速度，而增加事务处理指令则是为了提高 COBOL 编译程序的编译效率及其目的程序的执行速度。还有些扩展则是用于提高操作系统的效率和质量。上述的扩展一般也只是使之要修改系统软件，主要是修改编译程序，如 PDP-11 系列在加了浮点运算指令后，可将 FORTRAN Ⅱ 改进为 FORTRAN Ⅱ-Plus；这些扩展一般是不会引起高级语言编写的应用软件不能兼容。

结合系列机的讲述，我们可以进一步看清从程序设计者看的系统结构和从机器设计者看的系统结构（其实就是从程序设计者看的系统结构的实现）的差别。同一个系列内的各挡机器都具有相同的从程序设计者看的系列结构（虽然有的挡可能有所扩展，但基本结构各挡都得相同），然而各挡机器可以采用不同的从机器设计者看的机器结构。例如，对图 1.18 的 IBM360/370 概念性结构，除了前面已讲过的可以采用不同的数据通路宽度外，对低挡的机器还可以采用把通道合并到中央处理机、分时执行的结构；对于中央处理机，可以采用指令的分析与指令的执行不重迭、顺序进行的结构，也可采用重迭、流水或其它并行处理结构。然而，这些都是实现相同的从程序设计者看的系列结构。

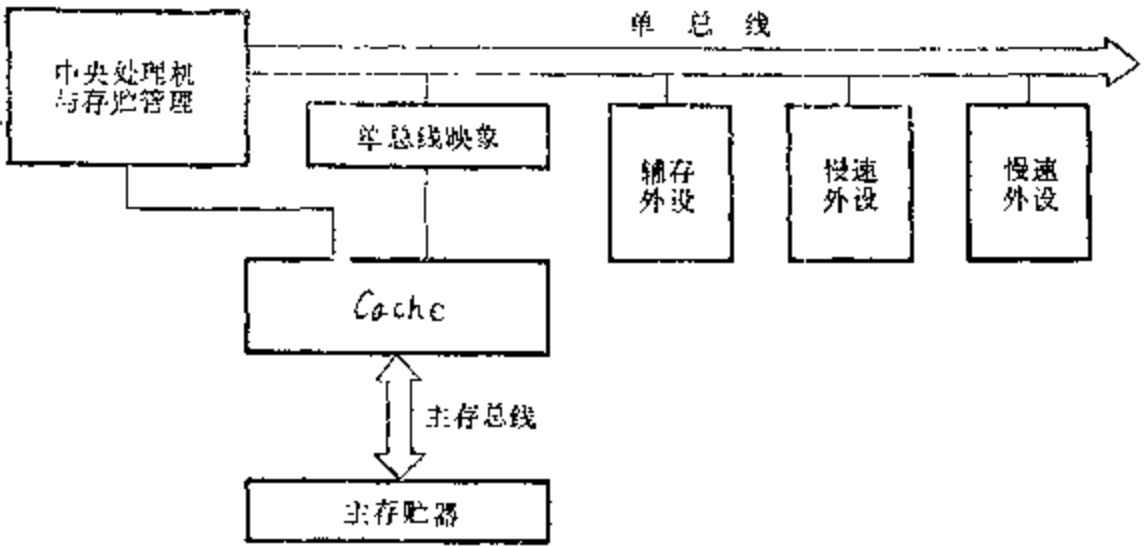
下面结合 PDP-11 系列的总线结构来进一步分析这二种结构的差别。其实，PDP-11 的单总线结构并不属于从程序设计者看的系列结构，因为 PDP-11 系列实际采用了多种总线结构方式，其中三种如图 1.21 所示。然而，它们都能通过访问主存贮器的指定单元实现与 I/O 设备的通讯联系。程序设计者其实并不需要看到单总线，他所需看到的是各 I/O 设备的寄存

器占据（对应于）主存贮器的指定单元，即 I/O 设备寄存器是按主存单元地址编号，并能用访主存操作对它们进行读、写。这才是属于从程序设计者看到的系列结构，即程序设计者所看到的 I/O 联结与使用方式。至于图 1.21 的不同总线结构则是程序设计者所不需看到的，它们是从机器设计者的角度，为了不同的速度要求而设计出来的不同结构。机器设计者完全可以不断改进它，但任何改进都不应影响到从程序设计者看到的系列结构，不然就会破坏系列机所要求的软件兼容。

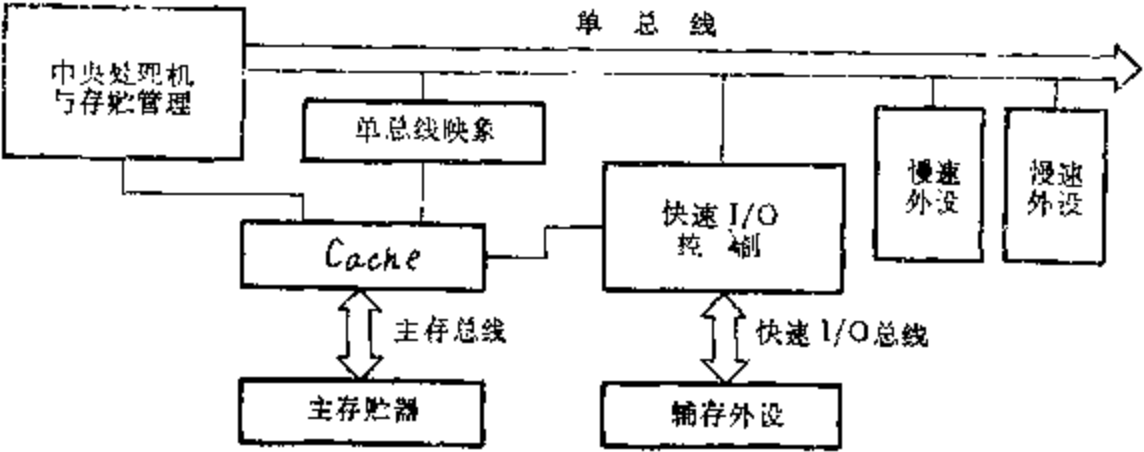
分清这二种结构很有意义，因为程序设计者看到的系列结构要求稳定，长期不变，以达到软件兼容，解决程序的可移植性要求，从而可不断地提高软件的质量，扩大其功能；但从机器设计者看的机器结构则要求随着器件技术的进展、市场状况和用户的不同需要而不断地



(a) PDP-11/04, 34 的总线结构



(b) PDP-11/44 的总线结构



(c) PDP-11/70 的总线结构

图 1.21 PDP 11 系列结构可用多种总线结构实现

改变。例如，DEC 公司就几乎每年出一种 PDP-11 的不同结构，如 75 年的 PDP-11/70 及 LSI-11 微处理器，76 年的 PDP-11/34，77 年的 PDP-11/60，78 年的 PDP-11/34C 和 VAX-11/780（它和 PDP-11 系列兼容，但字长和虚拟存贮地址由 PDP-11 的 16 位增加到

32位)，79年的PDP-11/44。80年PDP-11系列供应的品种是PDP-11/04, 34C, 44, 60, 70。看出，PDP-11系列是在不断地淘汰其性能价格比不适应市场发展需要的型号，如PDP-11/03, 20, 40, 45等等；这样，PDP-11作为一个系列，在保证向后兼容的前提下，通过不断地改进其各种速度的机器结构，在中小型机市场中长期占据主要位置。

在保持从程序设计者看的系列结构不变化的前提下，机器设计者仍然可以大有作为，不断研制新的机器结构。例如，Amdahl公司就是完全照搬IBM370的从程序设计者看的系列结构，使得可和370兼容，充分利用IBM370的软件；但又采用了新的机器结构和器件工艺，成功地研制出在性能价格比上超过370的Amdahl470。

〔通过系列机实现软件兼容确是解决软件要求环境稳定和机器结构要求迅速不断发展之间的矛盾的好办法〕六十年代以来，它对计算机工业的迅速发展确是起了很好的推动作用。然而，在软件兼容前提下，机器结构的发展毕竟是有局限的，往往难于采用完全新型的结构。显然，要求现在设计的机器要和十年前的机器兼容，那是很难实现结构上的突破。因此，从某种意义上说，软件兼容的要求已开始成为妨碍计算机系统结构更迅速发展的因素。但是，至今还没有更好的办法来解决程序的可移植问题。

至于不同系列间的程序移植，虽然可以采用模拟与仿真等办法，但其效率一般都较低。

3.1-3 模拟与仿真

前面讲的系列机办法所能实现的程序移植只能局限于具有相同系列结构的各机器之间，而如果要求程序能在具有不同结构（从程序设计者看的结构）的机器间相互移植，就得做到能在某一种系统结构之上实现另一种系统结构，即实现另一种机器的属性。从系统结构的主要方面——指令系统来看，就是要在一种系统结构上，实现另一种机器的指令系统，即另一种机器语言。在§1已经讲过，所谓高级语言就是在系统结构上通过编译程序实现的虚拟机语言。如果我们把上述另一种机器语言也作为虚拟机语言看待，那就可采用相似的层次结构来实现，如图1.22所示。图中，要求原在B机器系统运行的应用程序能移植于A机器系统，但A、B机器从程序设计者看的系统结构并不相同；因此，就需在A系统的机器语言级上，用虚拟机的概念实现B机器的指令系统。这里一般不是采用实现高级语言的那种翻译方法，而是采用§1讲过的解释的办法，即B机器的每一条机器指令是用一段A机器语言程序去解释。这样，就如同在A机器系统上也有着B机器的指令系统，从而使原先按B机器编制的应用程序可以移植、运行于A机器系统。

这种用机器语言程序解释实现移植的方法称为模拟（Simulation）。用这方法实现的B机器称为虚拟机（Virtual Machine），而实际有的A机器则称为宿主机（Host Machine）。为了模拟B机器系统结构的全貌，不能只是模拟机器语言，还得模拟B机器的存贮体系、I/O系统，甚至是控制台的操作，也就是说还得形成B机器的操作系统，才能使原按B机器系统编制的应用程序得以正确运行。这些模拟程序的编制都是很复杂、费时的。B虚拟机的操作系统需由A机器的操作系统控制（如图1.22所示），实际上是把它作为A机器系统的一道应用程序来看待，从而使得按A机器系统编制的应用程序和按B机器系统编制的应用程序可以在A机器系统上共同执行。显然，由于B机器的一条机器指令现在不是直接由硬件执行，而是需经多条A机器指令的软件模拟来实现，因此这种用模拟方法

实现的程序移植，其速度效率显著下降。所以，模拟法只适合于移植运行时间不长，使用次数少的程序，而且不能是在时间关系上有约束和限制的程序；显然，对实时中断与时钟中断的响应时间，B 虚拟机器与 B 实际机器当然不能一样。

虚拟机概念是多道程序技术和虚拟存贮技术的发展，它的主要用途不在于模拟移植，而在于实现多种操作系统的共同执行（例如，IBM 370 系统中通过虚拟机技术使得 360 的 OS/360 操作系统与 370 的 OS/VS2 操作系统可共同执行）以及使每个用户都能接触到传统机器级的所谓自虚拟系统等等。这些，已不属本书的讲述范围。

由图 1.22 可以看出，若 A 机器采用微程序控制（如图中所示），则一条 B 机器指令的执行其实是通过二重解释：先是经 A 机

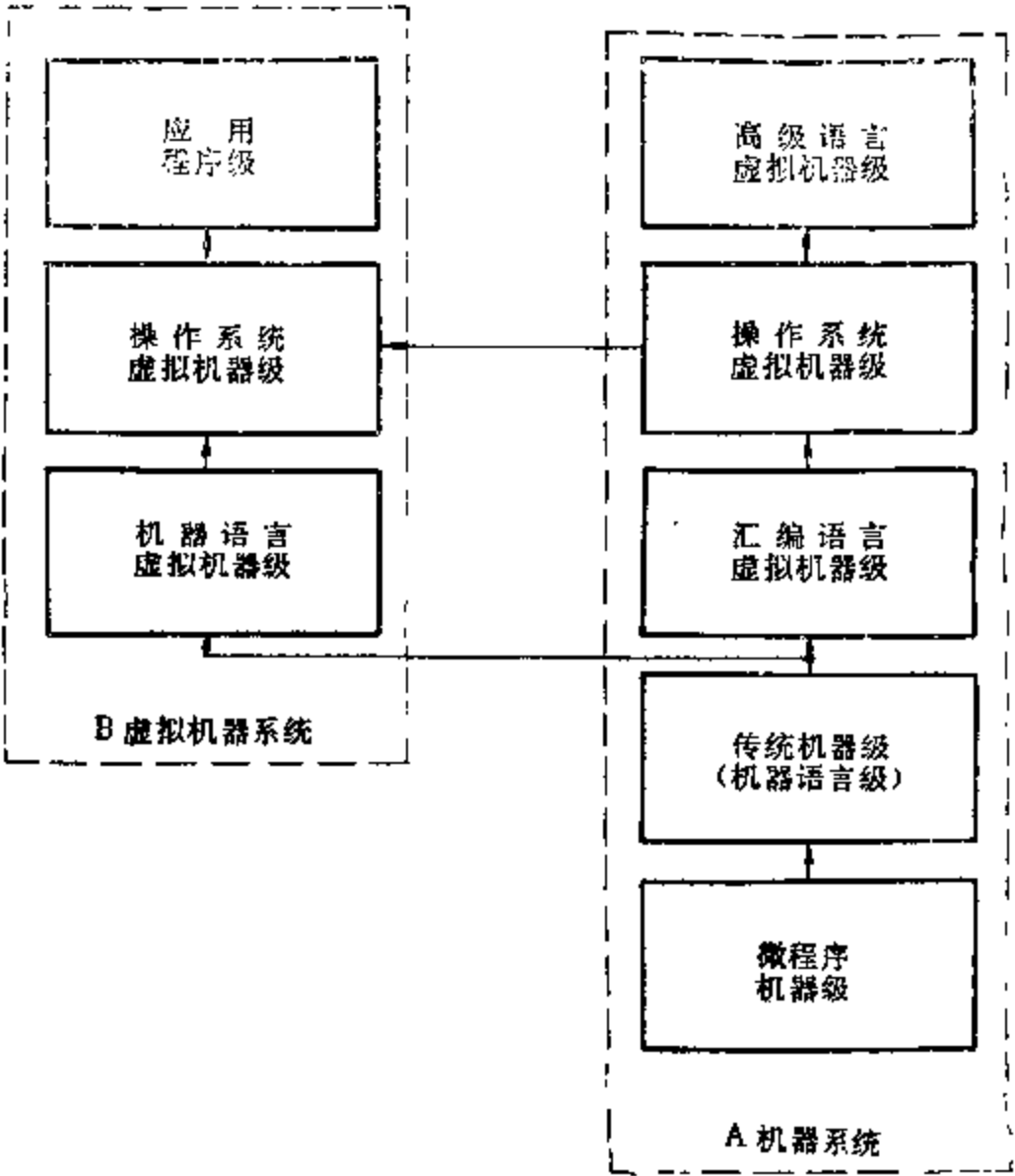


图 1.22 用模拟方法实现应用程序的移植

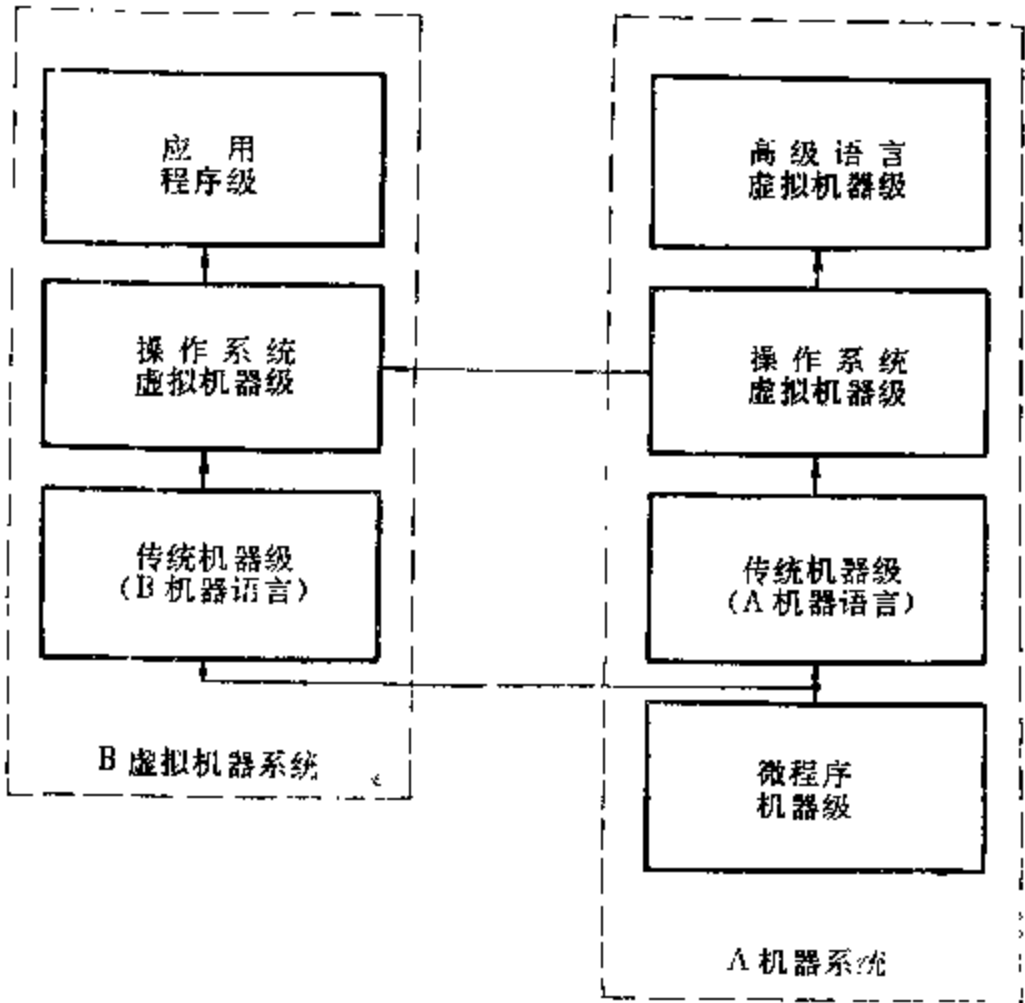


图 1.23 用仿真方法实现应用程序的移植

器语言程序解释，而后每条 A 机器指令又经一段微程序解释。显然，直接用微程序去解释 B 机器指令（如图 1.23 所示）是会加快解释过程。这种用微程序直接解释另一种机器的指令系统的方法称为仿真（Emulation）。A 机器也称为宿主机，被仿真的 B 机器称为目的机（Target Machine）。仿真与模拟的主要区别在于解释用的语言，仿真是用微程序解释，其解释程序存在控制（微程序）存贮器；而模拟是用机器语言程序解释，其解释程序存在主存。可以在一台微程序机器上实现多种机器语言的思路在 § 2.2-3 中已提过，这和仿真是同一概念。

如同模拟方法一样，除了仿真机器指令系统之外，还得仿真存贮体系、I/O 系统与控制台的操作等等。如果 B 机器从程序设计者看的系统结构与 A 机器的有较大差别，那只用微程序仿真是难于在 A 机器上构成它的，尤其是当其 I/O 系统结构差别较大时更是如此。

因此，在实际应用中，不同系列间程序移植往往是通过仿真和模拟二种方法的并用来实现。对于用得频繁的机器指令，尽可能用仿真方法来提高速度，对于用得较少，且用微程序仿真难以实现的某些指令及 I/O 系统等适于采用模拟方法。就是对系统结构差别不是过大的，也往往需用软件方法（即模拟方法）把 B 机器的存贮器和寄存器结构以及 I/O 联结方式映象到 A 机器的相应结构。例如，IBM 370/145 为此设置了仿真支持包 ESP 软件，用于实现对某些非 370 系列机器的程序移植。下面举 IBM1401 程序移植于 370/145 的例子。这里 370/145 是宿主机，1401 是虚拟机、目的机，如图 1.24 所示，在 370/145 的主存内有 370 的程序、1401 的程序、370 的磁盘操作系统 DOS 以及仿真支持包 ESP；在 370/145 的控制（微程序）存贮器内则存了 370/145 指令系统的解释微程序和 1401 指令系统的解释微

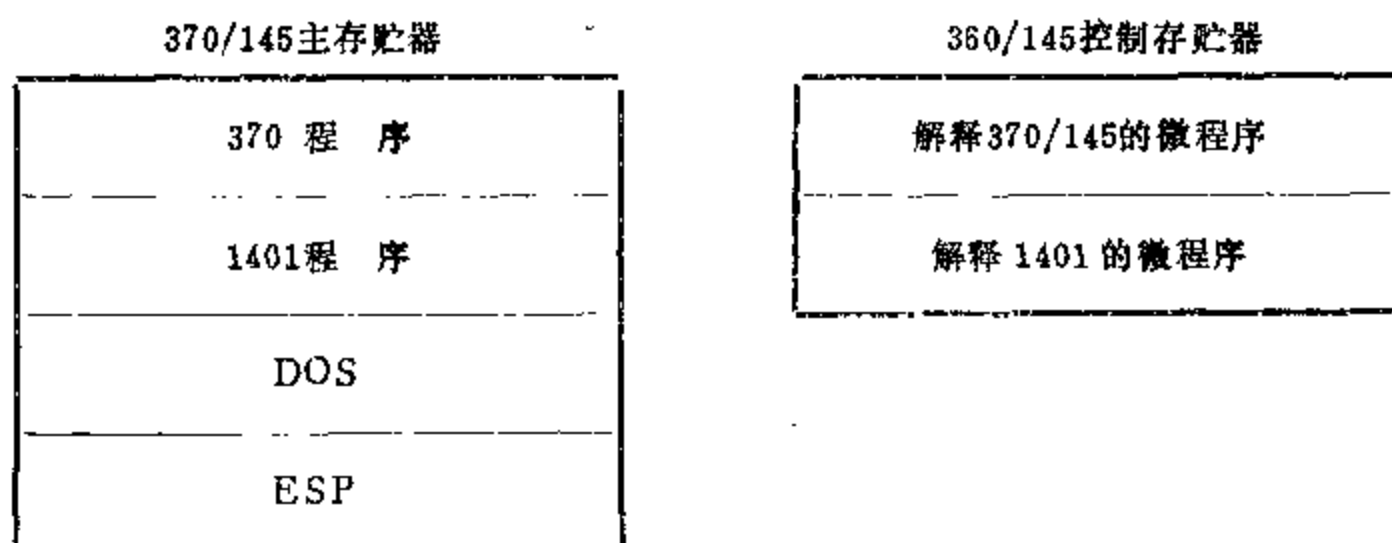


图 1.24 IBM1401程序移植于 IBM370/145

程序。370 还为此设置了 DIL（执行解释）机器指令，用于对移植来的程序（这里是 1401 程序）执行取指令并控制转入解释这条指令的微程序。370/145 由 DOS 控制，可使本身的 370 程序和移植来的 1401 程序共同执行。DOS 把 ESP 作为应用程序，当要执行 1401 程序时，先进入 ESP，把对应 1401 的起始状态及 1401 对 370/145 的相应映象置定好，而后通过执行 DIL 指令开始由 1401 程序取第一条指令并由相应微程序执行之；执行完后又经 DIL 转入执行第二条指令。并如此继续执行下去。看出，1401 程序的执行基本上是一个仿真的过程。在执行过程中，按 1401 的指令和数据格式，由主存的 1401 程序区取出指令与数据。当进行到需执行 I/O 操作时，一般就需转入模拟，即转入相应的 370 模拟子程序。

关于微程序仿真，我们再讲几句。在前面讲系列机时我们讲过，一种系列结构可以用多种机器结构实现。对于每一种机器结构，可以按系列结构的要求进行优化设计，使其最适应于这种系列结构的实现。然而，其逆过程却难以成立。对微程序控制机器来讲，也就是说，微程序机器级的结构是按优化实现（仿真）从程序设计者看的系统结构来设计的，但一旦设计好并制成后，在同一个微程序机器结构上却难于优化仿真差别大的另一种系统结构。这是因为微程序级的结构是深深地依赖于或取决于机器级的系统结构。结合图 1.23，就是说 A 机器系统的微程序机器结构是以优化仿真从程序设计者看的 A 机器的系统结构来设计的；但若 B 机器的系统结构和 A 机器的相差较大，那用 A 机器的微程序机器结构是难于优化仿真

B 机器的。联系上述宿主机和目的机概念，就是说，若 B 目的机的系统结构与 A 机器的相差较大，那就难于在 A 宿主机上优化仿真 B 目的机。

对所谓通用宿主机结构的研究就是企图研究出这样一种微程序机器结构，它能优化仿真范围尽可能广的系统结构。其实，位片式微处理器就是这种通用宿主机的雏型，器件厂生产微程序位片，整机厂可以对它配上不同的微程序以构成多种型号机器。然而，至今还没有研制出真正通用的宿主机结构，还有很多难题需要解决。看来，只从微程序技术着手难以突破，还应结合于优化系统结构的研究。

还要注意到，要真正推动仿真工作的开展，一定得研究微程序软件。目前实际在使用的只是微程序汇编语言及微程序模拟程序，这是很不够的，需要解决微程序高级语言及控制微程序在主存和控存间的调度和多个微程序在控存内的共同执行等的微程序操作系统。相信随着控存价格的进一步下降，控存容量的不断扩大以及仿真的进一步需要，微程序软件是一定会得到发展的。

就如何解决程序的移植问题，本节讲述了统一高级语言，系列机以及模拟与仿真等方法。系列机是当前普遍采用的好方法，但它只能实现同一系列内的软件兼容，而兼容的约束往往是限制系统结构取得突破的重要因素；统一的高级语言，是重要的方向，是我们要努力去探索研究的，然而，语言的标准化问题，如同计算机领域内其它方面的标准化问题一样，是大家都感到很需要但又是很难解决的，模拟方法灵活性最大，从原理上讲它可以实现程序在任何机器间的相互移植，然而其效率很低，速度损失很大；仿真方法在速度上的损失要小得多，但目前只是在系统结构差别不大的机器之间适于采用它，不然效率也会降低，而且还得加上模拟方法才能真正实现。不过，对仿真的研究，其意义主要不在于为了实现程序的移植，它对推动计算机系统结构在理论和实践上的进展很有价值。

系统结构设计者在设计他的机器时，一定要把程序的移植要求牢牢记在心中，只有这样设计出来的机器才能有生命力，才能得到推广使用。

关于软件对系统结构的影响，我们这里只讲了程序的可移植性要求的影响。显然，高级语言如何实现也应对系统结构的设计有重大影响。就是说，系统结构的设计本应更多地考虑如何优化高级语言的实现，如何更好地面向高级语言等等。这些，我们在第二章再来讲述。同样，可以看成是机器系统结构引伸的操作系统必然也会对系统结构的设计和发展有很大影响。关于这一点，我们也放在第二章来讲述。

3.2 应用对系统结构的影响

计算机应用对系统结构的发展当然有着很重要的影响。在前面所讲的内容中有些实际上就是和应用的要求有关。例如，系统结构的设计应适应程序可移植性要求，应考虑为用户提供更好的性能价格比等等。此外，如何便于使用也是应用要求中的重要一环。这包括从操作台上着手，在设计开关、指示器时如何使得一般使用者都能正确地使用机器；当使用错误时，能发出相应的信息提醒并帮助使用者进行纠正。还要注意减少命令的种类和简化操作步骤，例如把使用类型归并成几类，只需按下对应某一类的按钮（功能键），就能发出相应的一串操作命令，而不必每一个基本操作命令都由使用者发出。此外，还应为使用者能方便地纠正、撤消其原先已发出的命令提供更好的硬件支持等等。

各种应用的这些共同要求还可举出许多，但本节主要讲系统结构如何为各种不同的应用

提供相应的支持。

关于这点,有些是明显的。举例来说,有的应用(如导弹、飞行器上用)要求高可靠性,在系统结构上就需要采用第六章要讲到的各种容错技术;有的应用(如学校用机器等),对可靠性的要求不那么苛刻,就不必采取过多的可靠性措施。这是按不同的可靠性要求对应用进行分类以便系统结构为之提供相应的支持。那么,从系统结构观点,应该如何分类,才便于适应各种应用的要求呢?

3.2-1 多功能通用机概念

在前面我们讲过,四十年代末期出现的 Von Neumann 型机器是为计算弹道、解偏微分方程而设计的。但很快就发现它也适合于算很多其它的题目,从而称之为通用计算机。就是说,对五十年代初期的计算机应用领域来讲,它是适应于各种用途的需要。但是,从五十年代中、末期以后,计算机的应用范围逐步扩大,开始由只用于科学计算而扩展到商业、事务处理。后者的特点是输入、输出量大,计算量却不大,而且要求有十进制运算及字符行处理等。这时,就出现了一批专用于商业、事务处理的机器,它的系统结构在这方面的处理能力较强;然而由于没有浮点运算,没有快速的二进制加、减、乘、除能力等,所以其科学计算能力就比面向科学计算的机器差。我们可以把这种机器称之为商业、事务处理专用机。而当机器的应用领域也扩大到过程控制(如武器控制、工业生产控制等),则出现了面向各种控制用途的专用机,其系统结构特点是具有实时处理能力(如中断处理能力强)和与外界的多种接口(如数——模转换接口及采样处理等)。

然而,用户总是希望它的机器的应用范围能愈宽愈好。因为,科学计算用机器(如安装于计算中心的机器)在某些时候可能要用于统计计算,它就希望有事务处理用的字符行处理和十进制运算能力;进一步,发现字符行处理指令在编译程序的设计中也很有用处。至于控制专用机的那些结构特点,如中断处理能力,也是机器分时、多道操作(计算中心机器就要求能分时、多道操作)所必需的,而且科研用机器有时也需能处理实验所得的物理量,也要用到控制专用机的那些与外界接口和相应的处理能力。同样,控制用机器有时也希望能兼作工程计算用或作统计计算用。而且,用户还希望当其机器的应用领域改变时,不必重新购买机器。这样,到六十年代初、中期,用户就愈来愈要求能把科学计算、事务处理和实时控制这三方面的结构特点合并在一台机器之内。

另一方面,六十年代中期以前,计算机厂家、机器设计者在设计事务处理专用机、控制专用机和提高科学计算用机器的性能的实践中也逐步摸索到了这三方面应用的结构特点及其相互关系。而且,随着器件的价格和失效率的降低以及硬件技术(如微程序技术)的发展,使得有可能在价格不变的情况下提高机器所具有的功能,使之可以同时具备这三方面的结构特征;虽然这样一来,机器所用器件数会增加,但其平均无故障时间(二次故障间的平均时间 MTBF)不仅不会下降,还有所提高。

这样,从六十年代中期以后,IBM360 等一批面向多种用途的多功能通用机相继问世。IBM360 以微程序技术实现了从当时来看是庞大的指令系统,它包括有科学计算用的相当完整的浮点运算指令、事务处理用的字符行处理指令、十进制运算指令以及某些实时控制用指令。它具有灵活的输入、输出结构(参看图 1.18);通道的个数可以根据不同的输入、输出量和不同的辅存容量而改变,每个通道所带的外部设备个数也可增减;而且,只要外设控

制器具有 I/O 接口总线所约定的接口标准，那由它控制的各种外设都可接上。它所能带的外设从典型的输入、输出设备：磁鼓、磁带、磁盘，直至模——数转换设备和各种通讯外设。它具有灵活的多级中断系统，能适应实时控制和分时、多道程序等的需要。它的主存容量也能根据需要而改变，而且只要在规定的最小存贮容量之上，不同的主存容量并不影响机器以及操作系统的正常运行。再加上这种通用机具有速度不同的多挡机器，并采用 § 3.1-2 所述的系列机概念，因此用户根据应用的不同只需选购相应的那挡机器并配置相应的外设和相应容量的主存即可。

在这种通用系统结构之上又配上了批处理、分时和实时等操作系统，就使得同一种型号的计算机系统更能适用于从科学计算到实时控制的各种不同应用。

上述由最初的简单通用机变化到出现多种专用机，又进而发展到面向多种用途的多功能通用系列机，是计算机工业逐步成熟的重要标志之一。由于机器的型号得以减少，而且同一型号的适应面宽，又便于扩展，使得每种型号的生产量可以增大、生产寿命可以延长（即不会老是要改型）。因此，计算机厂家就有可能下功夫于提高质量、提高可靠性和降低价格。这反过来又扩大了机器的应用范围，增加了机器的销售量。而且，由于型号相对稳定，就有可能积累使用经验，尤其是积累更多的软件，使计算机的功能进一步加强。

相反地，只为某种用途，甚至只为某种设备研制的专用机，由于其适应面窄、产量少，难于积累生产经验，难于采用投资大的先进生产工艺。从而其质量和可靠性提高慢，尤其是价格降不下来，阻碍了机器的推广使用，反过来也使得计算机工业因销售量不大而难以发展。当然，我们这里指的是在通常环境（体积、环境温度）下工作的专用机。至于是在特殊环境（如弹上、水下等）下工作的专用机，当然是必不可少的。

多功能通用机的概念起始于大、中型机，后来小型机以致微型机也逐步多功能通用化。小型机开始时是用途较窄的，或是应用于 OEM（初始设备制造）作为某个设备的控制环节，或是作为辅助大型机用，帮助它作些预处理。后来，其功能逐步扩大与多样化，而且其价格并没上升，因此适应多用途的小型机的用途日益广泛。“多功能通用”性能下移到小型机是符合前面图 1.13 所示的情况。为了保持小型机价格便宜的特点，不仅是外设数量、类型可以选购；而且，有些运算部件，如浮点运算部件、字符行、十进制运算部件也可选购。这样，科研计算用户就可选购浮点运算部件，而作事务处理用的就可选购字符行、十进制运算部件。这从数据表示来看，可以看成是选购各种数据表示类型，如选购浮点运算部件则相当于在系统结构内有了浮点数据表示等等。这种通过为不同的用途提供相应选购件使同一型号机器能适应多用途要求的办法是可取的，这种思路近年来有了进一步的发展，我们稍后就要讲到。

可以说，多功能通用系列机的出现并得到推广是推动计算机工业得以迅速发展的重要原因之一，至今仍然是计算机厂家和用户所普遍接受的办法，也是设计和选择系统结构所应遵循的重要原则之一。可惜的是到了七十年代我国才开始接受这种思路。

3.2-2 吸收专用机系统结构的成果

那么，是否在有了多功能通用系列机之后，就再没有，也不需要研制面向某种用途机器的系统结构呢？不是的。因为对每种计算机应用的新领域或是某个原有领域，应用要求的提高都必然要求研制优化于这种应用的系统结构。而且有的应用（如数据库），用现有通用

机器是很难达到高效率甚至是无法实现的。然而各种单用途专用机由于其适应面窄，利用率不高，难以得到较大发展。因此问题不在于要不要研制针对某种用途的机器，要不要研究这种用途的系统结构，而在于如何最快地把研制某种新用途机器系统结构的经验和成果转移到通用机的系统结构上，使得通用机的功能和性能得以不断提高，从而使通用机也能适用于这种新用途。因为，就是对这种新用途也往往是在能用得上大批量生产的通用机时才能真正得到推广。事实上，从有了多功能通用机之后，这种不断吸收专用机系统结构成果的情况才不断出现，下面举几个例子来说明。

例如，在六十年代末期，为某些要求高可靠性的机器研究了较多的容错技术，如汉明纠错码，不只是在磁盘、磁带上采用了，而且在磁心存贮器和数据传送过程中也采用了。这些技术接着就在 IBM360 多功能通用机的改进型 IBM370 上得到采用，从而提高了多功能通用机的性能，并使它能适用于要求高可靠性的一般用途。当然，随着计算机应用范围的扩大，对可靠性的要求也在不断提高，七十年代就研制了具有更高可靠性的机器，这不只是满足了某些要求可靠性很高的用途的需要，而且使容错技术大大推进了一步。我们相信这些容错技术及其进一步的发展中那些被证明是成功、且不难实现的部份定会应用于八十年代的通用机上。

又如，从不断提高运算速度来适应科学计算的要求，在七十年代初、中期研制成了阵列机和向量机等高速机器，如 ILLIAC—IV 等。它的价格昂贵，不是一般用户能买得起的，而且使用复杂。然而，在研制和使用这些机器的实践中所积累的经验所取得的成果却已被通用机采用来提高其性能。

先是 IBM 为其 370 通用机提供阵列运算部件，它以外设设备形式接到已有的通用机。输入、输出操作及文件管理等由通用机执行，而阵列部件则专用于对阵列型数据快速计算。

而后，到七十年代末期有更多的公司出售阵列部件，它们的性能又进一步提高，而且可以接到诸如 PDP-11 等小型机上，用 CALL 语句调用，部件本身还配有实时管理程序。

这样，就阵列部件本身来看，它实际上是功能很强的专用处理机，然而它又是作为外设接到通用机上来提高其性能。以这种方式把专用机成果应用于通用机，使通用机能满足某种用途的需要是可取的，因为这不影响到通用机原有的兼容性。阵列处理部件的加入对用户程序可以是透明的，用户程序不必因这种部件的加入而需要重新编写。程序仍由主机（通用机）执行，只是当某些功能适于由阵列处理部件完成时，才把它们由主机转去快速执行。

在八十年代，非数值运算的应用，主要是数据管理方面的应用将会获得很大的发展。它会要求有很大的数据库（例如超过 10^{10} 字节），若用现有的通用机系统存这种数据库，则用现有的索引方法根本不可能实现快速查询或修改。因为光靠软件方法不只是速度满足不了要求，而且会使软件非常复杂，因此必须借助于能提高机器智能程度的各种专用硬件。另外，还得靠硬的方法实现对信息的保护。这样，近年来对数据库专用机的要求愈来愈强烈，有不少人正致力于这方面的研究。VLSI 的发展又为数据库专用机的研究提供了可能性。

由于现有机器的系统结构基本上是按顺序执行和集中处理设计的，但数据库所要求的快速查询却希望有并行操作和分布处理。这里，智能磁盘控制器是一个方向。它是在现有磁盘控制器之内再加上半导体快速缓冲存贮器和处理器，使得原由中央处理机对磁盘信息进行的查找、表格处理和数据块的分解等转为由磁盘控制器就地直接执行。处理器可以是高速微处理器或是专为上述目的设计的、并行能力强的 VLSI 片子。由于所有磁盘控制器都有这种

智能，因此这种分布处理系统能并行地同时对所有磁盘的信息进行分析、查找，其速度当然比起逐个地把各个磁盘信息送往中央处理机去处理要快得多。

可以予见到，随着数据库机器的发展，它的成果会很快地被通用机系统结构所吸收，使得它的数据管理能力大大提高。

从上面的分析看出，多功能通用机目前基本上能满足各种应用的要求；当然，它的性能仍需不断提高，而不断地吸收为新用途研制的专用机的系统结构的成果是一个重要的途径。

本节就软件和应用如何影响系统结构的发展进行了一些分析，目的是要阐述为什么系统结构设计者不能只从硬件本身的需要（如提高运算速度）来设计和研究，而应充分考虑程序设计者和用户的需要，怎样去软、硬结合地提高解题速度与解题能力。由这节的分析还可看出，由于受诸如程序可移植性要求和复杂系统软件等的约束，系统结构的进展是难以有突变性的飞跳，一般是渐进的、积累的，至少从近期来看是如此。

§ 4 器件的发展对系统结构的影响

三十年来，器件的发展是推动计算机发展的主要动力，也是最活跃的因素。今后，依然如此。它对系统结构的发展也起着关键性的作用。当然，系统结构的发展反过来也对器件提出新的要求，促进其更迅速地发展。为了说明器件的发展对系统结构的影响，本节首先从系统结构的观点很简要地回顾器件的发展，并对器件进行分类；接着谈谈这些发展对逻辑设计方法的影响，虽然本书并不涉及逻辑设计的内容，然而器件发展对系统结构的影响是与逻辑设计方法的变化有关的，所以在本节中也少许讲讲；最后叙述器件的发展如何影响系统结构的发展。

4.1 器件发展的简要回顾

4.1-1 器件性能价格比的指数上升

器件从电子管、晶体管、集成电路直至大规模、超大规模集成电路的迅速发展是大家都知道的。计算机由采用电子管到集成电路作为基本元件用了 18 年时间，而进一步进展到意义更大的 LSI 则只化了 8 年时间。

器件速度的迅速提高使得每级门延迟 t_d 由 55 年的几百毫微秒缩短到 78 年的 3 毫微秒（指 78 年出售的机器中已装上了 $t_d = 3$ 毫微秒的器件），呈指数地下降，如图 1.25 所示。由图可见，随着集成电路工艺的成熟和完善，不同速度机器所用组件的 t_d 差别已趋缩小。至于主存速度，在过去十年内提高了一个数量级。到 80 年，采用 MOS 工艺的存储器，其存取周期最快的已接近于 100 毫微秒，而采用双极型工艺的则已能接近于 10 毫微秒。

器件集成度的提高也是指数型的。图 1.26 是每片逻辑器件的门数随年份的变化曲线。

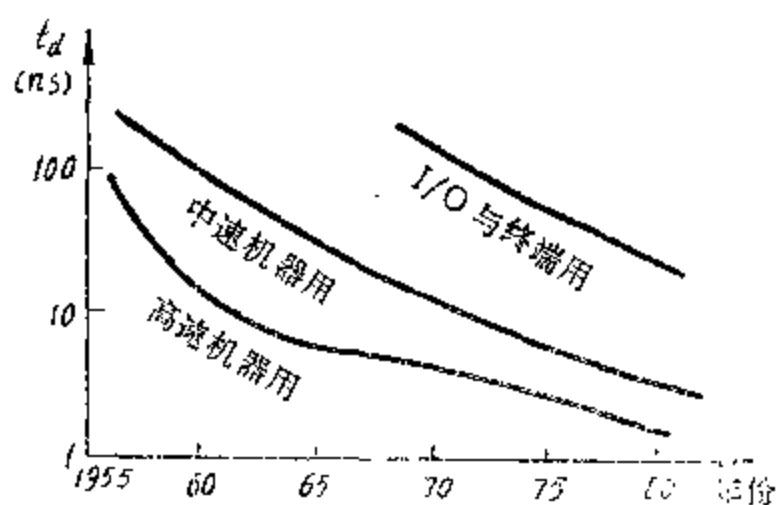


图 1.25 每级门延迟 t_d 随年份的变化

半导体存储器片子的集成度，即便是随机访问存储器 RAM，每 2 年也可以提高一倍，从而使得自 53 年以来一直占据主存位置的磁心存储器，到 74 年开始被半导体存储器所取代。

从体积来看，同样功能的逻辑结构，80 年的只是 55 年（采用电子管）的 1/500；存储器的变化更大，80 年的只是 59 年（采用磁心）的 1/1600，是 53 年（采用磁鼓）的 1/6400。到七十年代末期，在 1/4 吋见方的基片上，已可集成 16K 位的 RAM，或 65K 位的 CCD（电荷耦合器件），或 100K 位的磁泡。

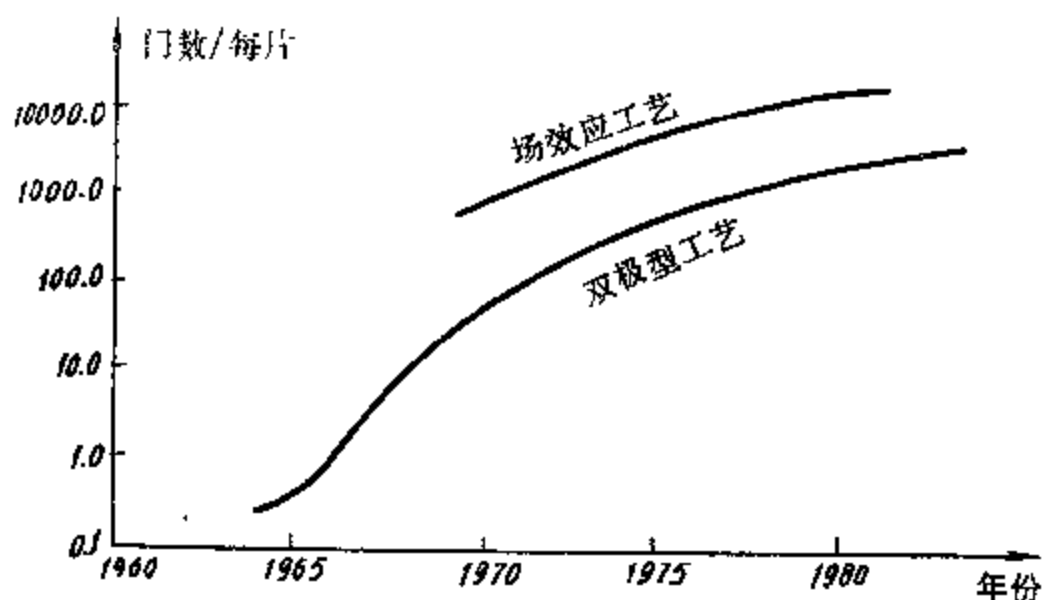


图 1.26 逻辑器件集成度的变化

从可靠性来看，由 63 年到 78 年，器件的失效率下降了二个数量级，使每个元件的平均无故障时间（MTBF）达到 10^7 小时以上。

总之，器件的基本性能，诸如集成度（或称之为密度）、速度、可靠性等的提高都是接近指数地上升，再加上电子计算机用的器件的价格随时间而指数下降，这就使得它们的性能价格比的提高特别显著。图 1.27 (a), (b) 是美国每个门、存储器件每位的价格下降曲线。正是有了器件在性能价格比上如此迅速的提高作为基础，计算机才得以有今天如此巨大的发展。

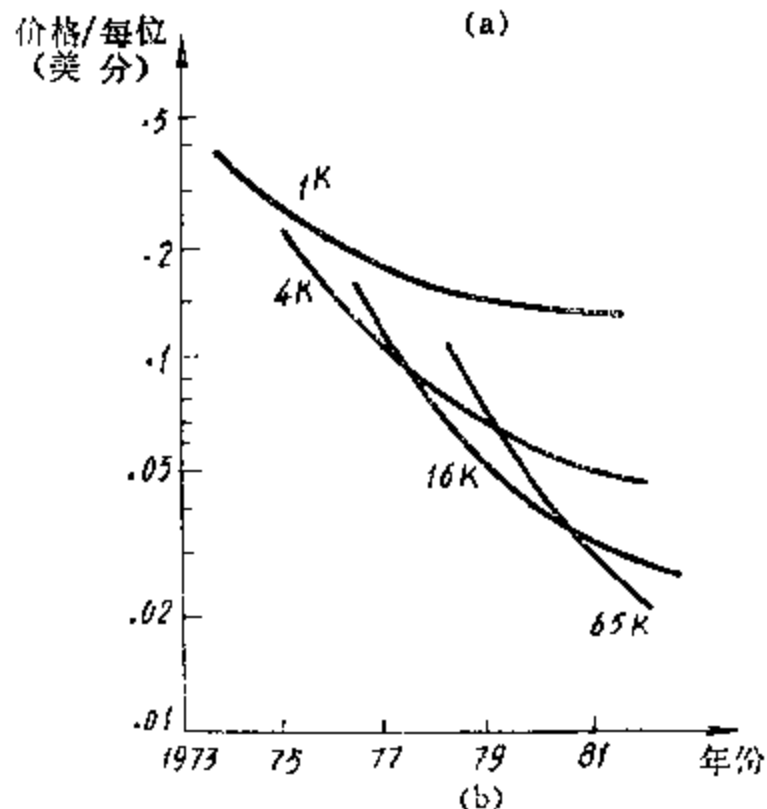
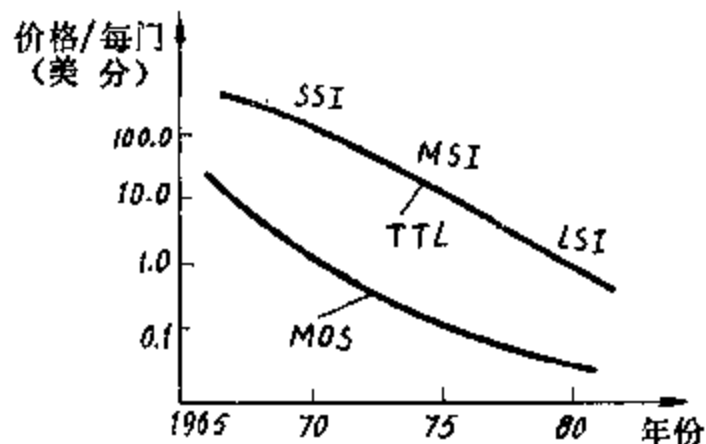


图 1.27 (a) 逻辑器件每个门价格下降曲线
(b) 存储器件每位价格下降曲线

4.1-2 非用户片、现场片与用户片

为了要讲述器件的发展如何影响到计算机的结构，先得讲述器件技术的发展使得器件的功能和使用方法发生了什么变化。大家知道，原先的器件，其功能是完全由器件厂在生产时定死了的；后来，机器设计者就希望能根据需要来改变器件内部的功能以使它更能适合于机器设计者所设计的系统结构，而随着器件技术的发展逐步提供了这种可能性。下面，我们就按器件用户（机器设计者）能否更改组件内的功能来对器件进行分类。这可以分成用户不能更改型（简称非用户型）、

用户现场可更改型（简称现场型或半用户型）和器件厂按用户要求设计、生产的用户设计型（简称用户型或全用户型）三种。这种分法对结构设计是有意义的，下面分别来叙述。

非用户型组件是从数字集成电路开始出现就有了，开始是简单的门和触发器等，后来发展到多路开关、加法器、寄存器和计数器，以及存贮器等。这些都是组成计算机的基本单元，也就是说这种组件是可以通用于各种机器及其系统结构的。

然而，随着器件集成度的提高，每个片子内的功能应如何扩大和发展呢？对存贮器片子当然是不断扩大其每片位数，这种扩大丝毫不影响其通用性，因为机器设计者几乎完全不必修改其设计就能得到集成度提高和价格下降的好处。由于存贮器片子的位数增加一倍，只需地址码增加1位，即只需多用一个引出腿，这对封装及装配的影响都不大，所以，存贮器片子很适应于集成度的不断提高。

但是，对逻辑组件，集成度的提高有可能会影响到它的通用性。对加法器、寄存器和计数器这类部件，集成度的提高可以反映成位数的增加。这是不会影响到它的通用性，不过其引脚数却会因此而要求成比例地增加。而大家知道，目前组件的引脚数由于封装技术等限制只能有几十，就是在八十年代恐怕也只有100左右。除了这类部件外，还可用高集成度技术构成某些运算器，如高速 16×16 位乘法器就是其中的一种，它的相乘时间只需300毫微秒，片子上集成了5000个门。由于几乎所有机器目前都有乘法运算，因此这种VLSI片子的通用性也较高。然而，若用高集成度技术去构成某些强功能的专用片子，如外设控制器片子或接口片子，那它的通用性就要差，这是因为目前的接口及外设控制器都还没有标准化，它们只能通用于一个系列，至多通用于一个厂家内。对硬联主控制器更是如此，若用一片或多片VLSI来实现，则这种片子简直毫无通用性。因为就是同一系列内的各档机器其主控制器若用硬联方法实现，就几乎没有什么共同之处。然而，器件的销售量和其通用性是有着极大的关系。而只有销售量大的器件，其性能才可能不断提高，价格也才能不断下降。从国外情况看，组件的销售量增加一倍，其价格能下降25%左右。这样，当集成技术发展能够制成高集成度VLSI片子时，必须仔细分析用它构成何种功能的片子才能具有通用性。在八十年代的今天，所可能达到的集成度提高很快，预计到八十年代末，每片门数可达 $10^6 \sim 10^8$ ，这个通用性问题就越来越突出。

从七十年代中期以来，为了提高器件的通用性，开始出现用户现场可更改（可编）的片子。这种现场型或半用户片子的功能有简单的，有复杂的。PROM（可编程序只读存贮器）片子就是功能简单的这种片子中的一例，厂家只需生产一种全“1”的ROM片子，而用户可在现场，根据需要将它更改成任意模式的只读存贮器ROM。熔丝型PROM是大家所熟悉的。进一步的发展是FPLA（现场可编程序逻辑阵列），它的“与阵列”和“或阵列”的码点可由用户在现场更改。这二种片子都是存贮型片子，因为规整，所以更适于采用高集成度技术。FPLA的功能比PROM强，它可以更高效地代替硬联的组合网络（加上寄存器及反馈还可构成时序网络）。器件厂不必为不同的组合网络提供不同的片子，只需生产一种FPLA片子就能满足不同的应用。这样，这种现场型片子的通用性当然就比前述非用户片子的高。

从这种观点看，位片型微处理器也属现场型片子。因为器件厂生产的一种位片，由用户配以不同的微程序，就能具有不同的功能。就是说，同一种位片可以由用户构成不同指令系统的多种机器，这显然具有通用型片子的特征。从概念上来看，如果说PROM、FPLA是用硬的（如把熔丝熔断）方法进行现场更改，那位片型微处理器片子则是采用软的（通过编制不同的微程序）方法来进行现场更改。当然，微程序总得存放在某种存贮器中，存贮器内

容的更改可能是用硬的手段；然而，这只是软的方法的具体实现手段。

现场型片子不只可用硬方法，而且还可用软方法改变其功能的这种思路很有意义，它开阔了器件发展的路子，也给机器的设计方法提出了新的要求。其实，从这个观点看，就是单片型微处理器在某种情况下也可属现场型片子。这是因为微处理器不只是用于构成微型机，而且还用于实现所谓程序逻辑，即用微处理器配上相应的程序去代替原由硬联逻辑网络完成逻辑功能。之所以能够替代是因为软件和硬件在功能上是等效的，硬件的功能总是可以用一段程序去模拟，问题只是在于效率如何，在于哪种逻辑关系宜于用程序逻辑实现。当然，我们不能把程序逻辑当成是万能的。在微处理器刚问世时，确是有人提出今后再不会有门和触发器等构成的硬联逻辑网络，都可以用程序逻辑替代。但后来表明并不如此，不只是微处理器的品种和型号不能象门、触发器、寄存器、计数器那样稳定下来，而是型号愈来愈多；而且程序逻辑的应用范围是较窄的，只是那些变量不多、时间关系复杂，但操作速度要求较低的地方才宜于采用。程序逻辑的速度较慢是明显的，例如 $A \wedge B$ 这个关系式，用硬联网络实现时，只需经一个与门的延迟时间，但用程序逻辑实现则需经“取 A”，“与 B”这二条指令的执行，当然慢多了。但是，对速度要求较低的地方，用程序逻辑实现还是可取的，因为对某些逻辑要求，它用的片子数（单片微处理器加上 PROM 片子或某些接口片子），要比用小规模集成电路（SSI）的门、触发器构成的逻辑网络少，甚至在价格上都可能还要便宜。这样，PDP-11 的智能终端 PDT-11 就经一个 8 位微处理器实现它与主机的接口，这个微处理器的程序不论对 PDP-11 的用户程序或是系统程序，它都是透明的。当然，当单片微处理器用于构成微型机时，它应属于强功能非用户通用片。

由以上的简要分析看出，为了使器件厂生产的高集成度片子能有更大的通用性，能适应于更多种用途，生产现场型片子是一种有效的途径。然而，现有的现场型片子毕竟有局限性，不论 PROM，FPLA 或各种微处理器都只适于替代用低集成度的门、触发器等组件构成的逻辑网络中的一部分。因此，机器设计者就希望器件厂能根据他的要求生产适用于他所设计的机器的专用 VLSI 片子，即所谓用户片。用户片当然没有上述通用性。由于 VLSI 的设计、试制费用远高于生产费用，除非这种片子的用量很大，否则器件厂是不愿意生产用户片的（除非是整机厂自己有 VLSI 的生产线）。

近年来，为了减少用户片的设计、试制费，出现了一些折衷办法，例如门阵列片和门—触发器阵列片等。所谓门阵列片是指的器件厂生产规整的门阵列（即把门规整地按阵列排列），但各个门之间究竟如何联结可按用户的要求而定，因为这只不过影响到最后的一至二道工序。这样，因为片子的对外功能是按用户所提要求定的，所以对用户来讲是专用片；然而，对器件厂，它基本上仍然可按通用片生产，虽然硅片面积的利用率会降低。这种思路与早期的把单门或双门等低集成度组件装成通用插件，而从机架底板上通过不同的连线去实现各种逻辑要求（如构成六门触发器等）的那种方法是相似的。当然，门阵列的联结也不是任意的，它要受诸如引脚数等的限制。至于门—触发器阵列，其差别只在于阵列中不只有门还有触发器。由于这些阵列的规整性，它还是适合于实现高集成度的。

我们按非用户片、现场片（半用户型）和用户片这种分类方法来讲述，丝毫不是想减弱研究新功能器件的重要性。显然，某些新功能器件，如按内容访问存储器（CAM）片子，它的速度如果在八十年代能够迅速增高，价格能够明显下降，那必将对系统结构发生深远的影响。我们之所以如此讲述，是为了这样能便于讨论和理解器件的发展对机器设计的影响。

我们讲了器件的性能价格比是指数上升,其实,磁盘、磁带的性能价格比也是指数上升。例如,动头磁盘的存贮密度从60年的每平方英寸10K位猛增至80年的接近于每平方英寸10M位,而它的每位价格在七十年代平均每年下降22%。磁带也是这样,它的带速每年提高5%,而位密度每年增加25%。看来,在八十年代内继续为计算机采用的磁盘、磁带的这种发展速度将会保持。

4.2 器件的发展对逻辑设计方法的影响

器件的发展改变了并将进一步改变逻辑设计的传统方法。因为对于采用LSI、VLSI片子的计算机,逻辑设计的着眼点就不在于如何用最少的门数实现机器的系统结构,甚至不顾及单个门的延迟是多少,而是着重于分析采用什么样的组成方式更能发挥LSI、VLSI技术所应带来的好处,以及选用什么样的LSI、VLSI片子能使机器的性能价格比更好。就是说,逻辑化简已不是它的重要的一环。现在,评价逻辑设计的好坏,其标志已主要不是门数的多少,而是在实现系统结构所提出的功能和速度要求的前提下,设计时间的长短、系统效能的高低以及能否用得上大量生产的通用VLSI、LSI片子。

如果以逻辑化简作为主要的设计手段,斤斤计较于如何减少门数,致使设计时间拖长或是使得组成不规整,给诊断带来困难,这些都是不可取的。尽可能选用大量生产的高集成度通用片子是计算机组成设计中的重要一环。这里,有效的途径之一是如何更多地采用存贮逻辑。

采用存贮逻辑是指的用适于发挥高集成度技术、规整的现场型存贮器片子来实现原由硬联逻辑所完成的功能。这种思路是早在六十年代就有了,只不过随着集成度的发展,它日益被人们所重视。

微程序就是其中的一例。在§2.2-3微程序那节中已经讲过,微程序技术能把硬联控制器网络的复杂逻辑联结转变为控制存贮器的各种码点。在上一节还讲过,如果硬要把很不规整的控制器用VLSI技术实现,则这种VLSI片子很可能毫无通用性,亦即几乎没有生产价值。但用微程序方法则可采用前述现场型片子PROM来存放微程序,而这种片子的通用性很高,可以大量生产。

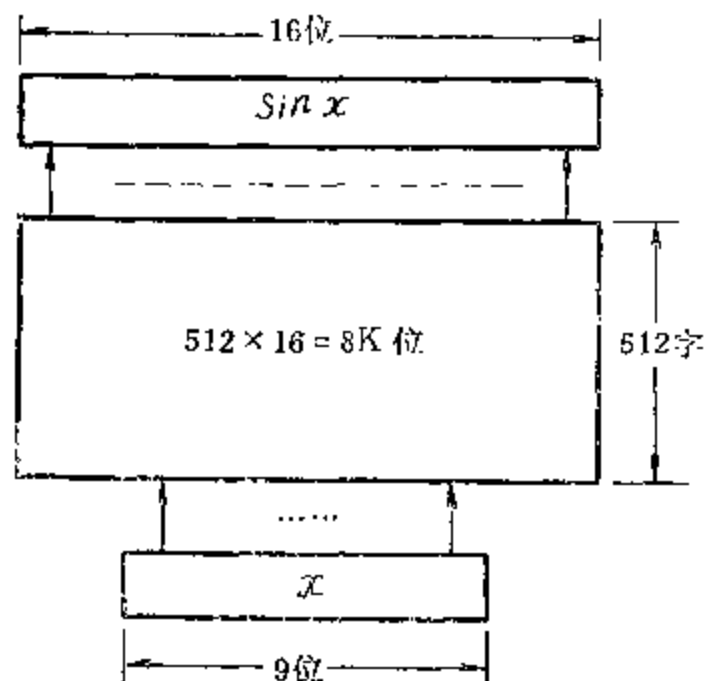


图 1.28 用查表法求 x 的正弦函数值

用查表法实现某些变换甚至运算是又一个例子。用查表法实现输入、输出码制的变换以及求三角函数值等的思路早就有了,只不过由于存贮器片子访问时间和价格的指数下降,才为它的实现提供了实际可能。例如,求从0度到360度的正弦函数值,为表示从0到360要用九位二进制表示,若函数值的精度要求很高,达16位,其所需存贮容量也只是 $360 \times 16 = 5.76K$ 位,因此,用如图1.28所示的 $512 \times 16 = 8K$ 位的存贮器片子是足够的。而单片8K容量的存贮器片子在七十年代末期就能制造了。由此例看出,虽然它只用了512字中的360个字,冗余了 $512 - 360 = 152$ 个单元。从传统的逻辑化简观点来看,因为它有“冗余”没消除,是个不好的方案;然而,从通用性观点看,只要

用的是大量生产的通用 PROM 存贮器片子, 这个方案还是合理的, 因为它反而能省钱。

其实, PLA 也是存贮逻辑中的一种, 它不只是“或阵列(对应存贮片子中的存贮阵列)”可编, 而且是“与阵列(对应存贮片子的译码器)”也是可编的。从而灵活性大, 并能以比存贮器少的“冗余”量去实现逻辑要求, 它的延迟时间已达几十毫微秒以内。它的入口宽度宽是很有用的特点, 包括 PDP-11 在内的好多机器都是把机器指令接到它的入口去形成对应的微程序入口地址。另外, IBM 的有些外设控制器也用 PLA 替代原来的硬联逻辑网络。

前述采用微处理器加 PROM 的程序逻辑也可算是存贮逻辑中的一种, 因为它的功能是由 PROM 中的内容(程序所)确定。

当然, 如何尽可能选用高集成度通用片子, 采用存贮逻辑只是许多方式中的一种。如果强功能的通用型非用户片能够满足设计要求, 那首先应选它, 因为它的使用要比现场型片子的简单。不过, 低功能的 SSI 片子(它们当然是通用型非用户片), 如门、触发器等, 在机器的有些地方总还是要用到的。

采用非用户片的设计方法, 大家是比较熟悉的, 而采用现场型片子时, 光用过去那种硬的逻辑设计方法已不够了。因为不论微程序的设计还是微处理器程序的设计都是用软的方法。虽然微程序语言往往是低级的汇编式语言, 但微程序高级语言的研究在七十年代一直在进行。逻辑设计中软、硬方法都得采用的状况今后还会发展, 就是硬的方法中今后也会更多的采用计算机辅助设计的软的方法。这些当然对机器设计者提出了更高的要求, 光有传统逻辑设计的知识是不能适应要求的。逻辑设计方法的这种变化也是软硬相互渗透, 相互影响的一种表现。

4.3 器件的发展是推动系统结构前进的重要因素

器件的发展是推动系统结构前进的重要因素, 或者说是关键因素, 过去是这样, 今后更是如此。

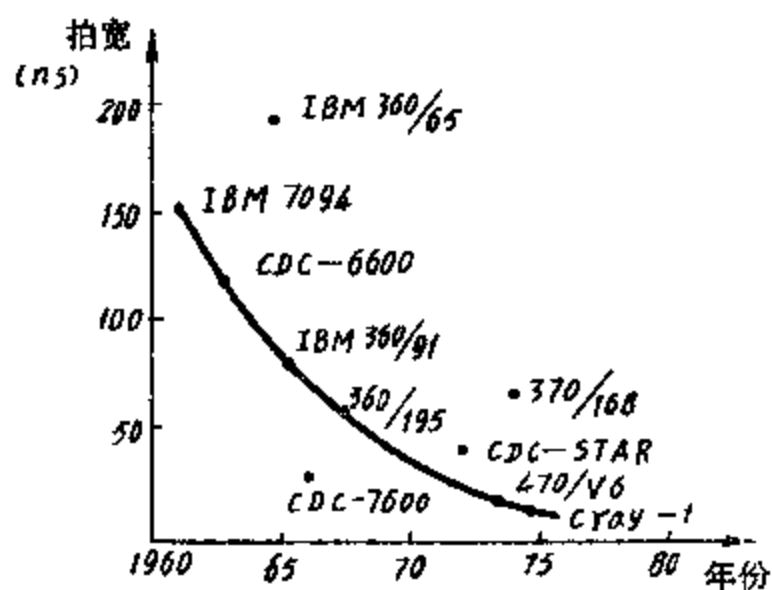


图 1.29 大型机拍宽的缩减曲线

由于器件速度的迅速提高, 使得机器的主频提高很快, 图 1.29 是大型机器节拍宽度(以下简称拍宽)的缩减曲线。虽然在七十年代内拍宽的缩短不显著, 但仍有可能在 1985 年前使拍宽缩短到几个毫微秒。七十年代内充分发挥 LSI、VLSI 的作用是使拍宽得以缩短的主要因素。例如, 在操作系统上和 IBM370 兼容的 Amdahl 470/V6, 由于很充分注意了这点, 使得它的拍宽比 IBM370/168 的要缩短一倍还多。因此, 如何把最长级数的环节尽可能

用最少的 VLSI 片子数实现, 是进一步缩短拍宽的重要途径。七十年代的大型机中, 最长级数的那部份的连线约只有 20% 是在片内。要使连线全在片内, 还得使集成度再提高 1~2 个数量级。

拍宽的缩短再加上系统结构上的改进使得机器的速度提高很快, 不过, 在七十年代提高的速度下降了。从 1955 年的电子管机器发展到 76 年的 LSI 机器, 解题速度提高了一百倍以上; 而且由于器件价格的下降, 机器的价格、解题费用也迅速下降。下表是这期间对于各

种混合的 1700 条指令的程序，其解题时间和解题费用的变化表。

年 份	器 件	解题时间 (秒)	解题费用 (美元)
1955	电子管	375	14.54
1960	晶体管	47	2.48
1974	SSI	5	0.28
1976	LSI	3	0.11

机器速度的提高大于主频的提高，这是由于这期间由计算机设计者看的系统结构的进展。然而，从根本上看，这些系统结构之所以能用得上，基础还在于器件的发展为此提供了可能性。这方面的例子很多。例如，第三章要讲到的流水技术需要用到大量的器件，如果器件的可靠性没有数量级的提高，那是没法采用的；又如第五章要讲到的能使机器解题速度得以显著提高的 Cache 存贮器概念及存贮层次，没有高速的、价格便宜的双极型存贮器件是根本不可能的；还有，微程序技术也是这样，这种思路是早在五十年代初期就已提出，然而真正得到广泛应用是在有了现场型 PROM 片子之后。

关于系统结构的“下移”，我们在 § 1.3-2 “系统结构与分型”中已经谈过，它是系统结构发展的重要特征。可以说，这主要取决于器件的发展，取决于器件性能价格比的提高。关于这点我们在前面已多次指出，而且其趋势是下移速度日益加快。这和 VLSI 设计工具的改进（采用更多的计算机辅助设计）使得设计费用下降、设计周期缩短是密切相关的。

例如 Cache 存贮器，早先只有大型机才有，由于它的效果显著，从而到七十年代末期甚至象 PDP-11/44，PDP-11/34 和 IBM4341 这样的中、小型机都已有了，而且其容量（PDP-11/44，IBM4341 都是 8K 字节）比早先大机器的还要大。在八十年代，由于器件厂已经能够提供包括有映象机构和控制机构的高功能 VLSI Cache 片子，势必将使更小的机器（从价格上来说）也能有 Cache 存贮器。

由于 VLSI 技术适合于产量大的产品，这正符合于系统结构的“下移”，因为“下移”就是把原先在大型机采用的，并已被证实是有用的新结构下移到销售量比大型机多的中、小型机去。由于所要下移的新结构的效能是肯定了的，而且又是用于中、小型机，需要量大；这样，器件厂就会愿意制造这种新结构的 VLSI 强功能片子。然而，当在大型机试用新结构时，其效能究竟如何是难于肯定的，因此大多是用通用的非用户片和现场片构成；所以其集成度反倒可能比下移于中、小型机的差。就是说，随着结构的下移是有可能出现新的强功能 VLSI 片子（前述 Cache VLSI 片子就是一例），这也是系统结构的发展反过来影响器件发展的一例。

由于器件的发展，系统结构中，不只是从计算机设计者看的系统结构下移很快，而且从程序设计者看的系统结构也是如此。例如在下一章要提到的，IBM 的操作系统用扩展部件，在七十年代中期只是 IBM370 中的大型机有，而到 70 年末期，则连 IBM4331 的小型机也可配上了。

随着现场片品种的增加、功能的加强以及用户片价格的下降，下移的速度必会加快，而且范围会更广。

我们说过,器件是发展分布处理的基础,正是由于有了VLSI技术的迅速发展,人们才预计到在八十年代分布处理必会有大的发展。

VLSI片子,首先是在中央处理机和主存内使用,从七十年代末期开始进入终端与I/O控制器。这种进入不应只是仍然维持原来功能不变,简单地替换低集成度的组件,而是应该通过VLSI的引入改变原先中央处理机与终端、I/O控制器之间的分工。就是说,应该通过VLSI的引入把原先由中央处理机实现的部分软件功能分散给终端和I/O控制器,使它们具有“智能”,为此,在终端内还得有运算、处理能力和存储器。由于微处理器和存储器片子价格的下降,增加这些功能已有了可能。

器件对系统结构的影响当然不只是上述的结构下移。其实,器件的发展还必然影响到算法和软件的发展。例如,由于微处理器价格的下降,从硬件结构上构成有几十、上百甚至上千个微处理器的并行处理系统已完全有可能,但障碍却在于至今还没有研究出合适的并行算法,这就迫使人们去发展新的算法。至于在软件方面,也因器件的发展而提出了很多新的观点。例如,随着中央处理机价格的进一步下降,分时系统的前景如何?又如,还提出应该设置软件检查机器(可由微处理器构成),把应用程序的检查和执行分布于不同的机器执行等等。

系统结构设计者必须密切注意器件的发展,并不断研究看这些新的器件和新的生产工艺究竟会给系统结构的发展带来什么样的新途径。

主要参考文献

- [1] A. S. Tanenbaum, "Structured Computer Organization," Prentice-Hall, 1976.
- [2] D. J. Kuck, "The Structure of Computers and Computations," Vol. 1, John Wiley & Sons, 1978.
- [3] H. S. Stone, (ed.), "Introduction to Computer Architecture," Second Edition, Science Research Associates, 1980.
- [4] C. G. Bell, "Computer Engineering-ADEC View of Hardware Systems Design," Digital Press, 1978.
- [5] G. J. Myers, "Advances in Computer Architecture," John Wiley & Sons, 1978.
- [6] G. G. Boulaye, "Computer Architecture," D. Reidel Publishing Company, 1977.
- [7] T. W. Pratt, "Programming Languages: Design and Implementation," Prentice-Hall, 1975.
- [8] A. K. Agrawala and T. G. Rauscher, "Foundations of Microprogramming: Architecture, Software, and Applications," Academic Press, 1976.
- [9] R. Hartenstein, "Increasing Hardware Complexity-A Challenge to Computer Architecture Education," The 1st Annual Symposium on Computer Architecture, 1973, pp. 201-206.
- [10] L. C. Wu, "VLSI and Mainframe Computers," COMPCON 78, Feb. 28-Mar. 3, pp. 26-29.
- [11] A. Recoque, "Survey of Main Trends in Computer Hardware Architecture," IFIP 80, Oct. 6-9, 1980, pp. 115-125.
- [12] M. J. Lynn, "Directions and Issues in Architecture and Language," Computer, Vol. 13, NO. 10, Oct. 1980, PP. 5-22
- [13] A. B. Salisbury, "A Study of General Purpose Microprogrammable Computer

- Architectures," Technical Report No.59, Digital Systems Laboratory, Stanford university, July 1973.
- [14] J. D. Bagley, "Microprogrammable Virtual Machines," Computer, Vol. 9, No.2, Feb. 1976, pp. 38-42.
- [15] E. I. Organick, "New Directions in Computer Systems Architecture," Euromicro Journal, Vol. 5, No.4, July 1979, pp. 190-202.
- [16] 苏东庄、李学干、任瑞英, "微程序工作的开展与通用微程序机器," 中小型电子计算机设计技术专辑, 1978年, 12-21页。
- [17] G. M. Amdahl, et al, "Architecture of the IBM System/360," IBM Journ. of R. and D., Vol. 8, No.2, April 1964, pp. 87-101.
- [18] J. E. Juliussen and W. J. Watson, "Problems of the 80's: Computer System Organization," The Oregon Report: Computing in the 1980's, 1978, pp. 14-23.
- [19] B. R. Borgerson and H. L. Apfelbaum, "Computer Systems in the 80's," The Oregon Report: Computing in the 1980's, 1978, pp.3-8.

第二章 指令与编址

本章首先讲述在计算机系统结构的设计中，确定机器数据表示的重要性及其分析，而后讲述编址方式和指令系统；最后讲述采用堆栈数据表示的机器的结构特点以及它与 Von Neumann 型机器的比较。

§1 数据表示

本节先是讲述数据结构与数据表示的关系以及确定数据表示的一些思路；接着分析最常用的数据表示之一，浮点数的表示范围及影响运算精度的因素；最后讲述加强系统结构对数据结构支持的两种措施：自定义数据表示与向量数据表示。

1.1 设计系统结构首先要确定数据表示

1.1-1 数据结构与数据表示

大家从“数据结构”课已学习到了计算机上常用的各种数据结构，如串、堆栈、队、向量、阵列、矩阵、链表、树和图等等。它们在各种计算机应用中要用到，也是程序的基本组成。然而，长期以来，实现这些数据结构的各种算法都几乎是立足于系统结构只提供按地址访问的一维线性存贮器以及最基本、最简单的数据表示。数据表示指的是能由硬件直接辨认的数据类型，如大家所熟知的定点数、浮点数和逻辑数等。这样，这些数据结构是要经软件映象，变换成存在按地址访问一维存贮器内的各种数据表示。如何能用最少的存贮空间存得下这些数据结构以及采用什么样的算法能最快、最简地存贮和访问到它们则是数据结构的研究课题。

显然，数据结构的如此实现，正是反映了三十年来以软件的复杂化为代价来达到硬件结构的简单化这样一种设计原则。

早先的机器只有定点数据表示和逻辑数据表示，相应的就只有定点运算指令（如加、减、乘、除、移位、比较等）和逻辑运算指令（如逻辑加、逻辑乘、按位相加、逻辑移位等）。那时，如果要想采用浮点数，就得用二个定点数表示，并用程序的方法把浮点数映象变换成机器的定点数据表示。显然，这是很不方便且效率很低的。因此，机器很快就设置了浮点数据表示，机器的指令系统相应地也就增加了浮点运算类指令（加、减、乘、除、比较、存、取等），并使运算器能直接对浮点数进行运算，即设置了浮点运算用硬件。而后，随着计算机的应用范围扩大到事务处理，就希望机器能直接处理十进制数，从而有的机器又有了十进制数据表示及相应的十进制运算类指令（如加、减、乘、除、移位、比较等），并使运算器能直接对二——十进制数进行运算；为此，或是设置专用的十进制数运算部件，或是使现有的定、浮点运算部件也能执行十进制运算。

至此，机器就有了逻辑数、定点数、浮点数和十进制数这些基本数据表示，而机器的运算类指令以及运算部件就是围绕它们设置的。就是说，机器的运算类指令以及运算器的基本

结构主要是按机器有什么样的数据表示来确定的。目前，直至小型机，甚至微型机的很多机器大都有了这些基本数据表示，因此，从大型机到微型机的运算类指令差别就不大。

当然，这些基本数据表示只是必不可少的、最基本的数据单元。它们只是上述那些数据结构的组成元素，光有它们并未能给这些数据结构的实现提供应有的更好支持。

显然，在系统结构的设计中应进一步考虑如何为数据结构的实现提供支持。变址操作是第一个这种支持，虽然它是1949年就已提出，且在1953年的Datatran机器上就有了，但直至今今天几乎所有机器中仍然都是不可缺少的。它直接支持了诸如向量这种线性表的实现，如图2.1所示，变址位指明变址寄存器号，改变变址寄存器值，由1到*l*，就能指向A向量中由1到*l*的所有元素，而指令中只要指明起始值就行。这样，同一条运算指令不用修改就能作用于整个向量。至于二维表，如二维矩阵，也可用变址操作实现，但需用二个变址寄存

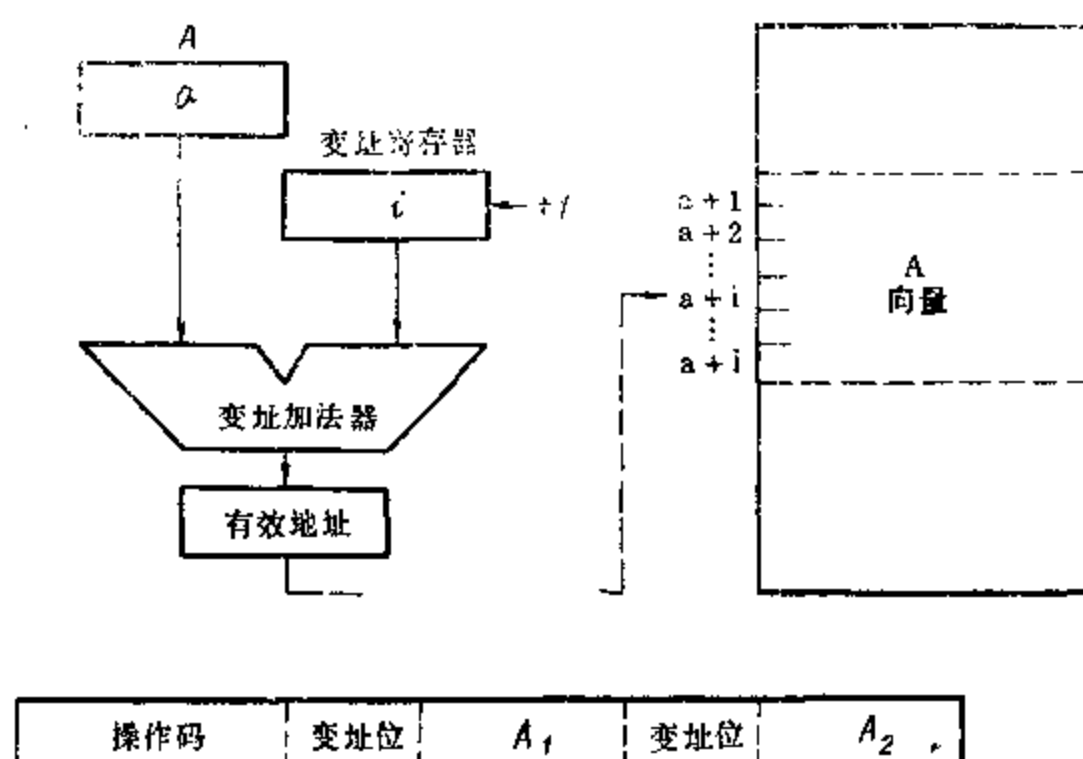


图 2.1 变址操作对数据结构的支持

器，分别表示A(i,j)的*i*和*j*，且有效地址的形成要复杂些。至于更多维的数据结构，用这种变址硬件去实现就不方便了。

变址操作的设置，使得不必对程序作任何修改就能用循环的办法对整个向量、阵列等进行运算，这对于实现程序的可再入性是有好处的。

从变址操作这个例子可以看出，系统结构应该也有可能为数据结构的实现提供支持。可变长字符串（行）数据表示的引入是又一个例子。大家知道，一个字符一般是单字节（8位二进制位）长，字符串的长度不宜固定，为了表示可变长字符串，需要指明首字符的地址以及字符串的长度（共有多少个字符）。随着字符串数据表示的设置，在指令系统中又增设了相应的字符串运算类指令（如联结、传送、比较、插入、编辑等等）。它们不仅用于输入、输出和事务处理，而且在编译过程中也用得上。这种表示是对串数据结构的有力支持。

因为要求指明各个字符的位置以及为适应按字节运算的发展，就必然要求除原有的按字编址（每个字有一个地址，可以按字的长度访问）外，还得有按字节编址。主存每个存储单元可能存得下多个（如1个或8个等）字节，但每个字节都是单独编址，都能单个被访问。进一步，由于PL/I等语言有位串数据结构，为了能对位串提供支持，从六十年代初开始，有

些机器还有按位编址。七十年代的大型机 STAR-100 和小型机 B-1700 都有按位编址。这样,随着数据表示的发展,机器内就同时有按字、按字节和按位的编址方式以及相应的寻址硬件。

在逻辑数、定点数、浮点数、十进制数、字符串等基本数据表示和变址操作的基础上,如何为数据结构提供更多的支持,这个问题日益被人们所重视,它是设计新系统结构的重要方向之一,其意义远比在指令系统中增加一条技巧性指令(如增加一种条件转移指令)要大。

如何从这方面缩短高级语言与机器语言的语义差距是从六十年代末以来一直受到重视的一个方向。

大家知道,二十多年来高级语言程序都是经编译程序翻译成机器语言程序后,通过机器执行得到结果的。编译过程可分为分析、综合二个过程,如图 2.2 所示。分析指的是对源程序的分析,主要包括词法分析、语法分析和语义分析,由它输出各种型式的中间语言程序。综合指的是目的程序的综合,它先对中间语言程序进行某种优化,接着经代码生成变成各种

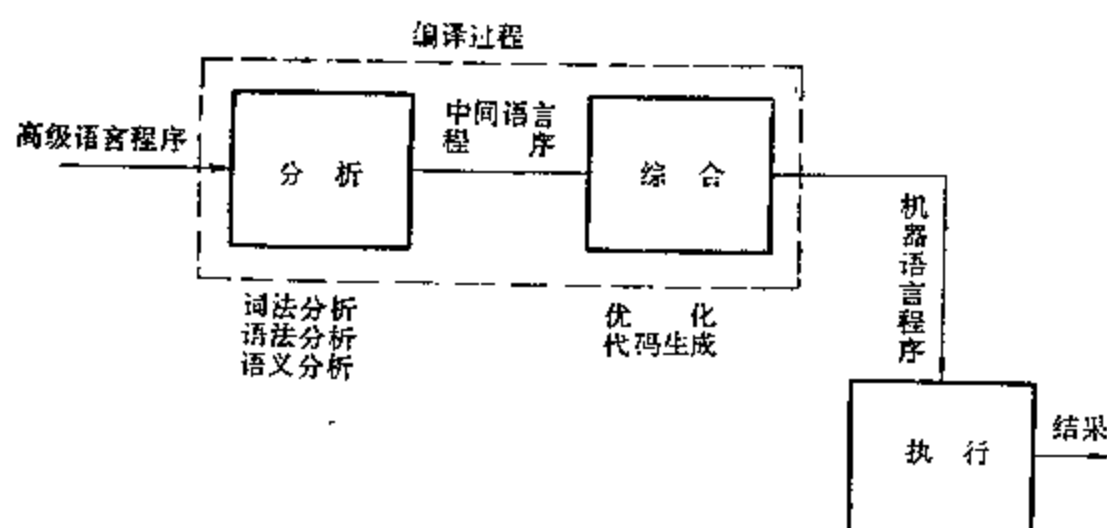


图 2.2 高级语言程序的编译

型式的目的程序,如汇编语言程序或机器语言程序等等。复杂的编译程序一般是用机器语言编制的,但机器语言通常并不是根据编译的需要设计,经编译形成的目的程序其效率一般是比较用手编的汇编语言程序或机器语言程序要差,主要表现为需用更大的存贮容量和更长的运算时间。因此,人们往往用编译时间和目的程序的执行时间以及目的程序所需的总位数来表征计算机系统实现高级语言的效率。

经编译实现高级语言就需在真正算题时间(即目的程序的执行时间)之外还要化费用于编译的机器时间,这部分时间所占比例并不小,有时甚至会超过目的程序的执行时间。之所以如此,很主要的一个原因在于机器语言并不是面向高级语言的实现,而是面向硬件实现的,这使得高级语言和机器语言在语言结构上有很大差别,尤其是在语义上的差别。人们往往以语义差别的大小作为衡量高级语言与从程序设计者看的系统结构在概念上相差的程度。这种差别表现在各个方面,数据结构和数据表示的差别就是一个方面。以运算符和数据类型的关系来看,那这二种语言的用法甚至是相反的。在高级语言中,是由说明语句指明数据的类型,使数据的类型直接与数据本身相联系,但其运算符却是通用的,并无数据类型的含义。例如,对 FORTRAN 语言,实数(浮点数) A, B 的相加是如此指明:

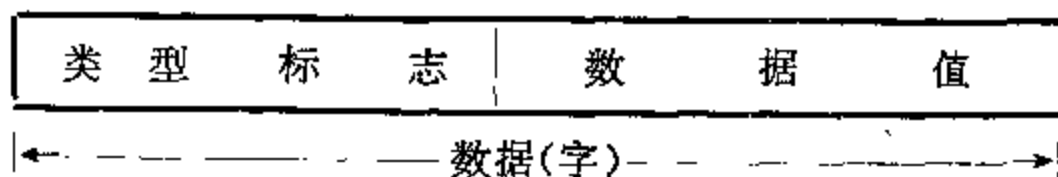
```
REAL A,B
A = A + B
```

它是先说明数据的类型，而后通用的“+”运算符按已给定的操作数类型执行实数相加运算。而机器语言却正好相反，它是由操作码指明的运算符决定操作数的类型，只要运算符是“浮点相加”，则不论操作数是否真是浮点数，恒按浮点数处理。即对于

$$A = A + B \quad (\text{此处“+”为浮点加法运算符})$$

不论 A, B 本来是何种类型，恒按浮点数进行相加。这样，在由编译程序形成目的程序的过程中，就需把高级语言程序中的数据类型说明语句变换成机器语言的不同数据类型操作，而且要保证运算符两边的操作数类型确是与运算符所要求的相符合。这些，都增加了编译的负担。

那么，能否使机器中的数据本身也带有类型特征呢？这就是说，使机器中的每个数据都带有表示数据类型的标志位：



由标志位指明这个数据是二进制整数、十进制整数、浮点数、字符串还是地址等。显然，这样一来，机器语言中的运算符也就和高级语言的一样，可以通用于各种数据类型，即加法指令只需有一种，而究竟执行的是浮点加还是定点加、字符加、十进制相加等则是由操作数的标志位所决定。这种带标志的数据表示法对简化代码生成有好处。例如，在代码形成中，当编译程序遇到“+”运算符时，本来是先要判定与它对应的操作数是何种类型，才能形成相应的相加机器指令，而现在，则可在遇到“+”时就直接形成通用的加法指令。这种带标志位的数据表示称为自定义数据表示，我们在下面 § 1.3-1 还要讨论。

至于对应常用的数据结构设置相应的硬件支持和数据表示自六十年代以后一直很受重视，并取得了很大进展。最初是六十年代对堆栈提供的支持，包括堆栈指示器和面向堆栈的指令直至在机器中设置硬堆栈。由于堆栈数据结构在编译过程和子程序调用中很有用，堆栈数据表示已在从大型机到微型机中广泛采用，并形成了面向堆栈的系统结构。这些，我们在 § 4 还要详述。

到了七十年代，在机器中设置阵列向量数据表示以及相应的运算指令和硬件以大大提高对向量、阵列数据结构的支持，是设计新系统结构的重要途径。这是由于高级语言程序中广泛使用阵列（数组）数据结构，包括多维阵列、相关型交叉阵列（即阵列中的每个元素又是一个子阵列）等，而且其它很多数据结构都可变换成用向量来描述。此外，高级语言的一个运算符能作用于整个阵列，并要求保证阵列的下标不应超出说明语句给定的范围。然而，包括 IBM370 在内的一般机器中，既无阵列型数据表示，又无对阵列运算的支持，至多是变址寄存器及其运算支持了对阵列中一行、一列的运算。这样，就得由编译程序，使用和阵列几乎无关的机器语言和数据表示去形成阵列数据结构并控制对它的运算，以及判断下标是否超出给定范围等。这些，必然严重地影响对阵列、向量数据结构的运算速度。因此，七十年代问世的巨型机，为了实现高速运算，几乎毫无例外地都设置了向量数据表示。看来，八十年代问世的通用机和系列机很可能会引用巨型机的成果，也设置向量数据表示及相应的快速运算硬件支持。

当然，系统结构缺乏对数据结构的支持还表现在其它方面。例如高级语言程序中，对记

录（非均匀性数据元素的集合）用的不少，但现有机器语言对此毫无支持。就是对字符串，目前绝大多数机器中，最多只有按字节（8位二进制位）编址的字符串数据表示，满足不少高级语言对字符串的使用，尤其难以实现对位串的处理。而且，目前机器指令中的字符串运算也只有简单的。因此高级语言中，诸如字符连接（相串），由字符串中取出指定的一段，对二个字符串进行相符比较等等都要用到编译程序。其实，即使是高级语言中用得普遍的实数，如 38.49，也需变换成二进制浮点表示后，才能由机器硬件去处理。这些都会使编译过程复杂化，使目的程序增长。还有，从根本上看，目前机器的存贮器是一维线性结构，但数据结构所要求的往往是多维离散结构，就是名称变量，其实大多不是线性顺序存贮，而是离散分布。这些，当然不利于数据结构的实现。因此，如何根据实现数据结构的需要来设计和改进系统结构应是我们很重视的一个方面。

综上所述，要设计系统结构首先要确定数据表示；而且，如何为数据结构提供更好更多的支持是设计由程序设计者看的系统结构的重要课题，它必然也决定着由计算机设计者看的系统结构。那么，应该遵循什么样的原则来确定机器的数据表示呢？

1.1-2 数据表示的确定

这是一个复杂的问题，这里只能原则地讲讲。

除了必不可少的基本数据表示之外，数据表示的确定是一个软、硬取舍的问题。这是因为各种数据结构本来是能够实现于只有最简单、最基本数据表示的机器中，只是在有了更好的数据表示之后，实现的效率可以更高罢了。

那么，某种数据表示的引入是否合适，其衡量的标准是什么呢？如同其它软、硬功能分配的分析一样，首先要看这能使实现时间以及所需存贮容量减少了多少。而对于实现时间是否减少的衡量，一个重要的指标是看主存与处理机间所需传送的信息量是否减少。下面以阵列数据表示为例来说明。设要对 A、B 二个 200×200 元素（定点数）的阵列进行相加运算，若用 PL/I 语言来编写，那很简单：

$$A = A + B$$

这个语句经 IBM370 的 PL/I 优化编译程序形成的目的码需用 6 条机器指令，其中有 4 条需循环执行 40,000 次。这样，就指令来看，在主存与处理机之间所需传送的字数为

$$4 \times 40,000 = 160,000$$

而就数据来看，由于每个循环需有二次取（取 A、B），一次存（存和），所需传送的信息量为

$$3 \times 40,000 = 120,000$$

总共需 280,000 个字的信息传送量。但若机器有阵列型数据表示，则只需一条机器指令就能执行二个阵列相加的运算，所以只需一次访存取指令操作；当然取、送数据的那 120,000 次还是需要的。由 280,000 个字的信息传送量减少成 120,000 个字，这当然会使实现时间减少很多。当然，高级数据表示的引入，往往伴随着在机器中采用能对这些数据表示进行快速运算的部件，由此而带来的实现时间的缩短在以后还要讲到。

由于数据表示的改进所带来的实现时间的缩短还表现为省了相当大量的辅助操作，下面还是结合阵列运算来说明。大家知道，对阵列数据结构，在运算时每次总是要判其下标是否是在所给的边界内，而这是相当费事的。例如，对

$$C(I,J) = A(I,J) + B(J,I)$$

这个 PL/I 阵列运算语句 (A、B、C 都是相同大小的定点数阵列)，如果不进行下标的越界检查，则只需 17 条 IBM370 机器指令 (共 62 个字节) 就可实现；但若需进行这种检查，则需 75 条机器指令 (共 274 个字节) 才能实现，而且就在没有越界时 also 需执行这 75 条中的 57 条。若机器有阵列数据表示，则为这种检查所需的硬件显然是很少的，而且可以和其它操作重迭进行，这样就会使阵列运算进一步加快。

衡量某种数据表示的引入是否合适的另一个出发点则是要看这种数据表示的通用性如何，利用率如何？对这个问题的分析要复杂得多，因为到了六十年代，尤其是到七十年代后，所要增加的数据表示已不是如浮点、字符串这些必不可少的、通用性大的简单数据表示，而是如堆栈、向量、阵列直至树和带标志位数据等比较复杂的数据表示。如果使机器的系统结构面向于一种高级数据结构，如有人提出的不是经指针实现树数据结构，而是具有树形数据表示的所谓树结构式机器，那对树数据结构来讲，它的实现效率当然很高，但对别的数据结构，那就可能较低。显然，若为此需化的硬件过多，那从性能价格比看是不合适的。就是对于比树数据表示要简单的堆栈、向量、阵列和带标志位数据表示，也存在这种通用性问题，例如堆栈型机器对于矩阵运算，效率就低；而阵列机则不只是有着如何使它能高效地解更多题目的通用性问题，而且还有阵列数据表示应能表示多大阵列的问题。如果能表示的阵列过大，则硬件设备量会过多，但利用率却可能会不高；如果能表示的阵列过小，则大多数阵列运算都要拆成多块，分批进入阵列运算部件，造成编译上的困难。

经过七十年代的实践和摸索，看来堆栈数据表示是已被肯定了，虽然光有它并不见得合适，而应是堆栈和通用寄存器并用；而带标志位数据表示和向量数据表示则是日益被系统结构设计者和程序设计者所接受，尤其是向量数据表示更是如此。所以，我们在下面对比较复杂的数据表示，就只讲这三种。

至于基本数据表示的确定，并不是什么问题都清楚和一致了。例如，对定、浮点数，目前大家已很习惯的定长表示不一定是好方式。因为，如果选定的字长过短，则能表示的数的范围过窄，对浮点数，若尾数过短，则精度会过低；而如果字长过长，则会造成存贮空间的浪费，即存贮器内有很多字的高位部分为“0”。这是因为程序中的数值分布不是均匀的，而且显著地集中于低值部分，即 0 和 1 值 (十进制值) 比例最高，而 1 到 10 值的出现概率又要比 50,001 到 50,010 的出现概率高得多。而且，定长表示还和某些语言的语义结构不一致。例如，COBOL 和 PL/I 语言就允许程序设计者使用可变长数据，而且是范围很宽的连续性可变，这样就势必要大大增加编译程序的负担，以解决如何把可变长数据映象成定长的数据表示。光是目前有些机器具有的多种定长数据表示，如 IBM370 的长字 (64 位) 和短字 (32 位) 二种，是适应不了可变长要求的；当然，要硬件结构能实现宽范围的可变长字，又能高速运算，其代价会是很高的。虽然按位编址是一种方法，但它只解决可变长字在主存中的存贮，而仍未解决可变长字的高速、高效运算问题。

我们当然不应反过来要求高级语言不准用可变长数据，因为高级语言既是和机器无关，它的字长本来就不应只能和所用机器具有的字长相对应。

因为是定字长，就更有着表示范围和表示精度的问题。对定长浮点数，尾数基值的大小对表示范围和精度有很大影响。目前有的机器的基值取为 2，有的则取 8，有的又取 16，并

不统一。另外，定长浮点数在运算过程中还会产生精度损失，这也是要注意分析的。因此，下面我们比较详细地讲述浮点数基值的选择和精度损失中的下溢处理问题。

1.2 浮点数基值的选择和下溢处理

1.2-1 浮点数基值的选择

大家知道，对于具有 n 位尾数的二进制定点分数来说，它的可表示值的范围为

$$2^{-n} \leq |N| \leq 1 - 2^{-n} \quad (2.1-1)$$

设二进制浮点数的阶为 p 位，尾数为 m 位，则其可能取的最小、最大值为：

$$\begin{aligned} |N|_{\min} &= 2^{-(2^p-1)} \cdot 2^{-m} \\ |N|_{\max} &= 2^{(2^p-1)} \cdot (1 - 2^{-m}) \end{aligned}$$

故其可表示值的范围为：

$$2^{-(2^p-1)} \cdot 2^{-m} \leq |N| \leq 2^{(2^p-1)} \cdot (1 - 2^{-m}) \quad (2.1-2)$$

对比 (2.1-1)，(2.1-2) 式，可以看出，浮点数比定点数有更大的表示范围。以上指的是基 $r=2$ 的情况。由于不同的基值对于浮点数的表示范围和其它参量有很大影响，为了便于对不同基值的分析，把 (2.1-2) 式改写成任意“基”的表示式：

$$r_m^{-(r_p^p-1)} \cdot r_m^{-m'} \leq |N| \leq r_m^{(r_p^p-1)} \cdot (1 - r_m^{-m'}) \quad (2.1-3)$$

式中 r_m 为尾数的“基”， r_p 为阶的“基”， m' 是以 r_m 为基的尾数位数。一般情况下，阶码都以 2 为基，所以上式可表示为

$$r_m^{-(2^p-1)} \cdot r_m^{-m'} \leq |N| \leq r_m^{(2^p-1)} \cdot (1 - r_m^{-m'}) \quad (2.1-4)$$

因此，关于不同基值的讨论，其实就归结为对不同 r_m 值的讨论。下面以 $r_m=2$ 和 16 的具体数值例子进行比较。这里，我们只比较正阶和正尾数，且都是规格化数，对于负阶和负尾数，结论也是适用的。

具有两位阶码和四位尾数的二进制浮点数见表 2.1。

从表 2.1 可以看出，最小尾数值为 $r_m^{-1} = \frac{1}{2}$ ，最大尾数值为 $1 - r_m^{-m} = 1 - 2^{-4} = 1 - \frac{1}{16} = \frac{15}{16}$ ，最大阶为 $2^p - 1 = 2^2 - 1 = 3$ ，可表示的最大数为 $(1 - r_m^{-m}) \cdot r_m^{(2^p-1)} = (1 - \frac{1}{16}) \cdot 2^3 = 7 \frac{1}{2}$ ，可表示的最小正浮点数为 $r_m^{-1} \cdot 2^0 = \frac{1}{2}$ ，可表示的尾数的个数为 8 个（由 $\frac{1}{2}$ 至 $\frac{15}{16}$ ），可表示的阶的个数为 4 个（由二进制 00 至 11），可表示的规格化数值的个数为 $2^p \cdot 2^m \cdot \frac{1}{2} = 32$ 个（由 $\frac{1}{2}$ 至 $7 \frac{1}{2}$ ）。

具有与上述二进制浮点数相同阶码和尾数位数的十六进制浮点数见表 2.2。

从表 2.2 可以看出，规格化十六进制尾数的最小值为 $r_m^{-m'} = \frac{1}{16}$ ，尾数的最大值为 $(1 -$

表 2.1 二进制浮点规格化数

	尾数位的权				尾 数 的 十 进 制 值	阶码 (共 2^p 个阶值)				阶 码 2^p 阶码
	2^{-1}	2^{-2}	2^{-3}	2^{-4}		00	01	10	11	
	1	2	4	8		1	2	4	8	
规格化 二 进 制 尾 数 值 (共 $2^m \times \frac{1}{2}$ 个 尾数值)	1	0	0	0	8/16	8/16	1	2	4	浮 点 数 值 (共 32个)
	1	0	0	1	9/16	9/16	$1\frac{1}{8}$	$2\frac{1}{4}$	$4\frac{1}{2}$	
	1	0	1	0	10/16	10/16	$1\frac{2}{8}$	$2\frac{2}{4}$	5	
	1	0	1	1	11/16	11/16	$1\frac{3}{8}$	$2\frac{3}{4}$	$5\frac{1}{2}$	
	1	1	0	0	12/16	12/16	$1\frac{4}{8}$	3	6	
	1	1	0	1	13/16	13/16	$1\frac{5}{8}$	$3\frac{1}{4}$	$6\frac{1}{2}$	
	1	1	1	0	14/16	14/16	$1\frac{6}{8}$	$3\frac{2}{4}$	7	
	1	1	1	1	15/16	15/16	$1\frac{7}{8}$	$3\frac{3}{4}$	$7\frac{1}{2}$	

$r_m^{-m'} = \frac{15}{16}$; 最大阶仍是 $2^p - 1 = 2^2 - 1 = 3$; 可表示的最大数为 $(1 - r_m^{-m'}) \cdot r_m^{(2^p - 1)}$
 $= \frac{15}{16} \cdot 16^3 = 3840$; 可表示的最小正浮点数为 $r_m^{-m'} \cdot r_m^0 = \frac{1}{16}$; 可表示的尾数的个数为
 15个(由 $\frac{1}{16}$ 至 $\frac{15}{16}$); 可表示的阶的个数仍为 4 个(由二进制 00 至 11); 可表示的规格化
 数值的个数为 $2^p \cdot r_m^{m'} \cdot \frac{15}{16} = 2^2 \cdot 16^1 \cdot \frac{15}{16} = 60$ 个。

对比表 2.1、表 2.2 可看出, 对于相同的阶码与尾数位数, 则尾数为十六进制的比尾数为二进制的有更大的数值表示范围。显然, 这个结论也适用于八进制和四进制与二进制的对比。换句话说, 对于大的 r_m 值, 为表示相同范围的数, 其阶码的位数 p 可以缩短。

还可看出, 尾数为十六进制的比尾数为二进制的有更多的可表示数的个数。可表示数的总数等于可表示阶的个数乘以可表示尾数的个数。由于 r_p 恒取 2, 所以可表示阶的个数恒是 2^p 个, 至于长度为 m 的尾数, 其可表示数的个数本应是 2^m , 但对规格化的浮点数来说, 可表示的尾数的个数就要减少。例如, 对尾数是二进制的, 尾数的最高位必需为 1 (前已讲过, 整个讨论只限于正阶正尾数), 因此可表示的尾数的个数就减少了一半, 而对尾数是十六进制的, 只要最高 4 位中有一位是“1”, 即为规格化的数, 因此可表示的尾数的个数减少了 $\frac{1}{16}$, 成为 $2^m \cdot \frac{15}{16}$ 个。同理可得八进制的可表示的尾数个数为 $2^m \cdot \frac{7}{8}$ 个。这样, 就可表示数的个数来讲, 十六进制的与二进制的比值为

表 2.2 十六进制浮点规格化数

	尾数位的权				尾数的十进值	阶码 (共 2 ⁰ 个阶值)				阶 码 16 ^{阶码}
	2 ⁻¹	2 ⁻²	2 ⁻³	2 ⁻⁴		00	01	10	11	
	16 ⁻¹					16 ⁰	16 ¹	16 ²	16 ³	
规格化	0	0	0	1	$\frac{1}{16}$	$\frac{1}{16}$	1	16	256	浮 点
	0	0	1	0	$\frac{2}{16}$	$\frac{2}{16}$	2	32	512	
	0	0	1	1	$\frac{3}{16}$	$\frac{3}{16}$	3	48	768	
	0	1	0	0	$\frac{4}{16}$	$\frac{4}{16}$	4	64	1024	
十六进	0	1	0	1	$\frac{5}{16}$	$\frac{5}{16}$	5	80	1280	数 值
	0	1	1	0	$\frac{6}{16}$	$\frac{6}{16}$	6	96	1536	
制尾数值	0	1	1	1	$\frac{7}{16}$	$\frac{7}{16}$	7	112	1792	(共 60 个)
	1	0	0	0	$\frac{8}{16}$	$\frac{8}{16}$	8	128	2048	
	1	0	0	1	$\frac{9}{16}$	$\frac{9}{16}$	9	144	2304	
	1	0	1	0	$\frac{10}{16}$	$\frac{10}{16}$	10	160	2560	
	1	0	1	1	$\frac{11}{16}$	$\frac{11}{16}$	11	176	2816	
	1	1	0	0	$\frac{12}{16}$	$\frac{12}{16}$	12	192	3072	
	1	1	0	1	$\frac{13}{16}$	$\frac{13}{16}$	13	208	3328	
	1	1	1	0	$\frac{14}{16}$	$\frac{14}{16}$	14	224	3584	
	1	1	1	1	$\frac{15}{16}$	$\frac{15}{16}$	15	240	3840	
	(共 2 ⁿ × $\frac{15}{16}$ 个 尾数值)									

$$\frac{2^p \cdot 2^m \cdot \frac{15}{16}}{2^p \cdot 2^m \cdot \frac{1}{2}} = 1.875$$

同理可得八进制的与二进制的比值为 1.75。

然而,尾数是十六进制的比尾数是二进制的可表示数在数轴上的分布较稀疏。例如在 $\frac{1}{2}$ 和 2 之间,二进制的分布有 15 个,而十六进制的只有 8 个,可见,要稀疏得多。为了进一步分析数值分布和基值的关系,这里,引入表示比 e 的概念。它指的是在相同 p, m 位数时,在 $r_m = 2$ 的最大值表示范围内,大 r_m 值的数值个数与 $r_m = 2$ 的数值个数之比。

已经分析过, $r_m = 2$ 时的可表示的规格化数的总个数为 $2^p \cdot 2^m \cdot \frac{1}{2} = 2^{p+m-1}$ 。又因为其最大阶值为 $2^p - 1$, 最大尾数值在 m 值较大时接近于 1, 所以能表示的最大浮点数值近似为 2^{2^p-1} , 下面就来求大 r_m 值时在 $r_m = 2$ 这个最大值范围内的浮点数个数。

设任意基 $r_m = \beta$, 我们总可以找到一个 q (不一定是整数), 使得

$$\beta^q \approx 2^{2^p-1} \quad (r_m = 2 \text{ 时的最大值}) \quad (2.1-5)$$

由前面讨论知,尾数以二进制表示时,只要尾数最高位是“1”,即为规格化数。而当尾数是十六进制表示时,只要尾数最高 4 位中有一位是“1”,也为规格化数。因此,我们可以推广到一般情况,当尾数是以任意基 r_m (或 β) 表示时,则只要最高 $\log_2 \beta$ 位中有一位是“1”,即为规格化数。

此外,对于 m 位尾数,当最高位为“1”时,其规格化数的个数为 $\frac{1}{2} \cdot 2^m = 2^{m-1}$ 个。次高位为“1”时,规格化数应为 2^{m-2} 个。同理,当最高第 $\log_2 \beta$ 位为“1”时,其规格化数应是 $2^{m-\log_2 \beta}$ 个。因此, m 位尾数以任意基 β 表示时,其规格化数的总个数应为

$$(2^{m-1} + 2^{m-2} + \dots + 2^{m-\log_2 \beta}) \text{ 个。}$$

对于二进制 p 位阶码,可表示阶的个数为 $(2^p - 1) + 1$ 。由于阶的基恒取 2, 所以,以任意基(尾数)表示的 (2.1-5) 式阶码的个数应是 $(q+1)$ 个。

这样,在 $\frac{1}{r_m}$ (即 $\frac{1}{\beta}$) 和 β^q 之间能表示的规格化数的个数为

$$\begin{aligned} & (2^{m-1} + 2^{m-2} + \dots + 2^{m-\log_2 \beta}) \cdot (q+1) \\ & = 2^m \cdot \left(1 - \frac{1}{\beta}\right) \cdot (q+1) \end{aligned} \quad (2.1-6)$$

注意,这里的 m 是尾数的二进制位数,而不是对应 $r_m = \beta$ 的尾数位数。这样,对正阶、正尾数,

$$\begin{aligned} e &= \frac{r_m = \beta \text{ 时, 在 } 2^{2^p-1} \text{ 范围内的浮点数个数}}{r_m = 2 \text{ 的浮点数个数}} \\ &\approx \frac{2^m \cdot \left(1 - \frac{1}{\beta}\right) \cdot (q+1)}{2^p \cdot 2^{m-1}} \end{aligned}$$

$$= \frac{2 \cdot \left(1 - \frac{1}{\beta}\right) \cdot (q+1)}{2^p} \quad (2.1-7)$$

由 (2.1-5) 得 $q \log_2 \beta = 2^p - 1$

将 $2^p = 1 + q \log_2 \beta$ 代入 (2.1-7)

所以,

$$e = \frac{2 \cdot \left(1 - \frac{1}{\beta}\right) \cdot (q+1)}{1 + q \log_2 \beta} \quad (2.1-8)$$

这样, 对于尾数是十六进制的, 若 $p = 8$ 位, 则 $q = \frac{2^p - 1}{\log_2 \beta} = \frac{2^8 - 1}{\log_2 16} \approx 2^6$, 所以表示比

$$e \approx \frac{2 \cdot \left(1 - \frac{1}{16}\right) \cdot (2^6 + 1)}{1 + 2^6 \log_2 16} \approx 0.47$$

前面已分析过, 对相同的 p, m 位, 十六进制的可表示数的个数是二进制的 1.875 倍, 这样, 对 $r_m = 16$ 及正阶来讲, 大约 $0.47/1.875 \approx \frac{1}{4}$ 的数值是在 $r_m = 2$ 的范围之内, 而有 $\frac{3}{4}$ 则是在 $r_m = 2$ 的范围之外。

由以上的分析可以看出, 基值愈大, 可表示的浮点数的范围愈宽, 而且可表示数的个数也愈多。然而, 在与 $r_m = 2$ 的浮点数相重迭的范围内, 数的密度分布却要稀得多。

另外, 从表示精度来看, 由于十六进制规格化数的最高位可能出现三个“0”, 这意味着对相同尾数位数 m 来说, $r_m = 2$ 的可能比 $r_m = 16$ 的有多三位的精度, 即若 $r_m = 2^k$, 则在最坏情况下, 尾数中只用到 $m - k + 1$ 位来表示, 看出, 浮点数的表示精度是随基值的增大而单调下降。

表 2.3 浮点加法移位距离与百分比

	二 进 制 尾 数		十 六 进 制 尾 数	
	移 位 距 离 (二进制位)	移位百分数	移 位 距 离 (十六进制位)	移位百分数
对 阶	0	32.64%	0	47.32%
	1~4	34.61%	1	26.02%
规格化	溢 出	19.65%	溢 出	5.5%
	0	59.38%	0	82.35%
	1~4	14.51%	1	7.24%

对于基值的选择, 还应考虑到对运算速度和精度损失的影响。对浮点加、减法运算, 对

阶和规格化都需进行移位。然而，尾数基值大的其移位次数要少得多，这可由表 2.3 看出。这个表是根据对大量指令执行的统计得出的，对十六进制的尾数，移位距离为 1 实质上是一次移 4 位二进制位。在对阶时，尾数为十六进制的，其不需移位的百分数比二进制的高得多，在规格化时更是如此，而且溢出（需右规）比例也小得多。显然，由于 $r_m = 16$ 的其移位的概率低得多，因此运算速度就得到提高。

另外，由于机器内的数据表示是定长的，因而右移（对阶及右规时）次数的减少，也会降低右移精度损失。从这些看，选择十六进制尾数有好处。

综上所述，选取大 r_m 值，对于扩大浮点数的表示范围和可表示数的个数以及减少右移次数，降低右移精度损失等有好处；然而，却会降低数据的表示精度并使数值的分布密度变稀。因此， r_m 值的选取是一种综合平衡，看来目前尚无定论。各个厂家的取值也不同，IBM370 取 $r_m = 16$ ；Burroughs 的大部分机器（包括 B-6700/7700）取 $r_m = 8$ ；而 PDP-11 和 CDC-6600，CYBER-70 等则取 $r_m = 2$ 。系统结构设计者在选取 r_m 时，需全面考虑，要知道不同 r_m 值的利弊，并采取相应措施减少弊害带来的影响。

1.2-2 浮点数的下溢处理

上面我们分析了在确定字长后，尾数基值的选择。提到了右移操作会把有效位右移掉而造成精度损失。精度损失主要来源于中间运算过程会使信息长度增长（如在相乘时以及在各种右移操作中），以致于超过运算器和存贮器的字长表示范围而不得不加以截断。当然，这可用两倍存贮器字来寄存信息和用双倍字长运算解决。然而，这都要花费两倍的存贮器空间和长得多的运算时间。因此，对一般的应用，应是着眼于如何采取措施尽可能地减少运算过程中带来的精度损失。这里，关键是要处理好尾数的下溢。

尾数下溢主要发生在加法过程中的对阶，右规移位以及在乘法过程中取单倍长度的结果乘积。常用的处理办法是：

(1) 截断

截断或称不舍入法，这是最简单、方便的办法。截断所带来的误差，在整数运算中最大可接近于 1（例如 7.999 截断到 7），在分数的二进制运算中接近于 2^{-m} 。它的误差曲线表示在图 2.2(a) 中。看出，误差都在理想曲线之下（都是负误差），只是那些重圆点处表示无误差截断。另外还可以看出，重圆点间的间隔几乎相等，这说明截断误差在不同的尾数值时，其概率分布是均匀的。

(2) 舍入法

采用这种办法时，误差曲线如图 2.2(b) 所示。由图可知，平均误差比截断法要小，但是稍偏于正误差（重圆点都在理想曲线之上），对十进制数，舍入相当于给最末位之后加 0.5，例如 $7.89 + 0.5 = 8.39$ ，即 7.89 舍入成 8，误差为正，而 $7.39 + 0.5 = 7.89$ ，即 7.39 舍入成 7，误差为负的，而 $7.50 + 0.5 = 8$ ，即 7.50 舍入成 8，误差最大。表 2.4 就表示了这种误差的可能分布。

从表中可以看出，误差有时是正的，有时是负的，但总的是稍偏于正误差。

(3) 恒置“1”法

所谓恒置“1”就是对被截断的尾数的最低位，无论原先是“0”还是“1”都恒置成

“1”。它和截断方法一样，不需要额外的处理时间，而舍入法在最坏情况下可能需要从尾

表 2.4 舍入误差及其偏差值

表示的数		舍入结果	误差
最末位	移出位		
1	$\frac{1}{2}$ $\frac{1}{4}$		
X	0 0	X	0
X	0 1	X	$-\frac{1}{4}$
X	1 0	X+1	$1 - \frac{1}{2} = \frac{1}{2}$
X	1 1	X+1	$1 - \frac{3}{4} = \frac{1}{4}$

偏差
为
 $\frac{1}{2}$

数最低位进位至最高位的进位时间。然而，它的最大误差值会比截断法的大，这是因为当尾数的移出位为全“0”，即出现 $10:000\dots0$ 时，恒置“1”法却置成 $11:000\dots0$ ，即产生尾数最低位“0”误为“1”的误差。

(4) ROM 或 PLA 舍入法

ROM 或 PLA 舍入法也称查表舍入法，它是用只读存储器或组合逻辑实现。它集中了上述各种下溢处理方法的优点，可以做到使平均误差近于零，而且也能基本避免上述舍入方法所需的相加进位传输时间。

它的工作原理是在 ROM 存储器中存有以下溢处理用表，表中每一个字的信息对应每种组合的下溢处理结果。一个 2^k 字 ROM 的 K 位地址对应尾数的最低 K-1 位和一位附加位；一般情况下，按舍入法编码的 ROM 输出即为尾数最低 K-1 位的下溢处理结果，如图 2.3 所示。但当尾数的最低 K-1 位为全“1”，时，则采用截断法来形成下溢处理结果，即

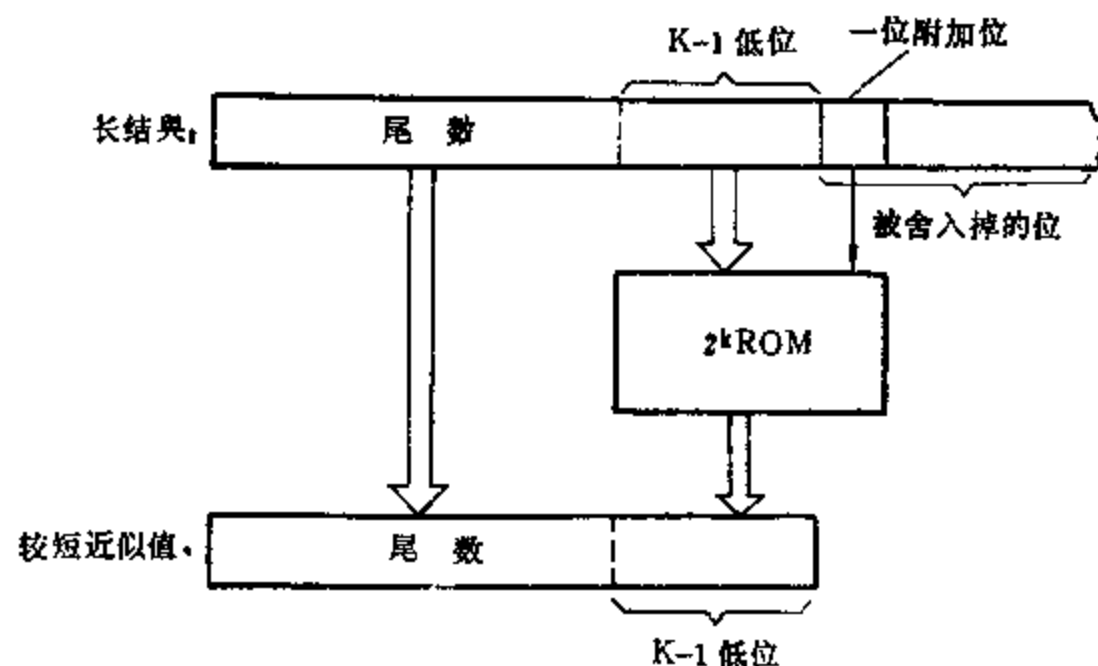
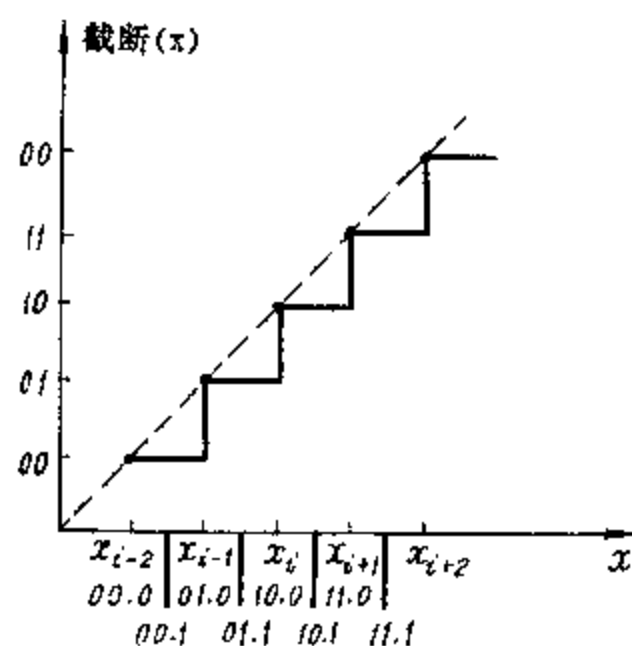


图 2.3 K 位 ROM 舍入

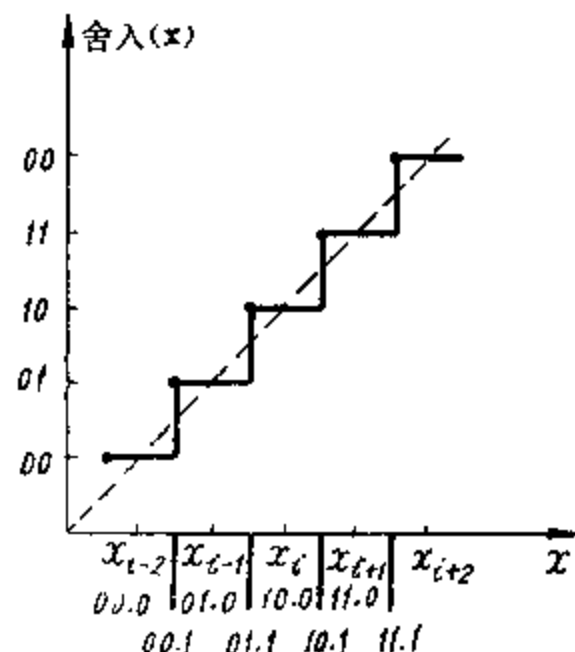
仍输出 K-1 位的全“1”，这样，截断所产生的负误差正好补偿了其它情况下采用舍入法所产生的正误差，使平均误差接近于零。而且，在 K-1 位为全“1”时，采用截断法，就避免了相加进位操作。可见，尾数最低 K-1 位的下溢处理结果，或是对应于截断法的应有结果，或是对应于舍入法的应有结果。

由于 ROM 的读出时间比加法时间短，所以这种查表法的处理速度比舍入法快。图 2.2 (d) 表示 8 字 ROM，K=3 的情况。看出，为使平均误差接近于零，按排成在 x_{i-1} 和 x_i 之

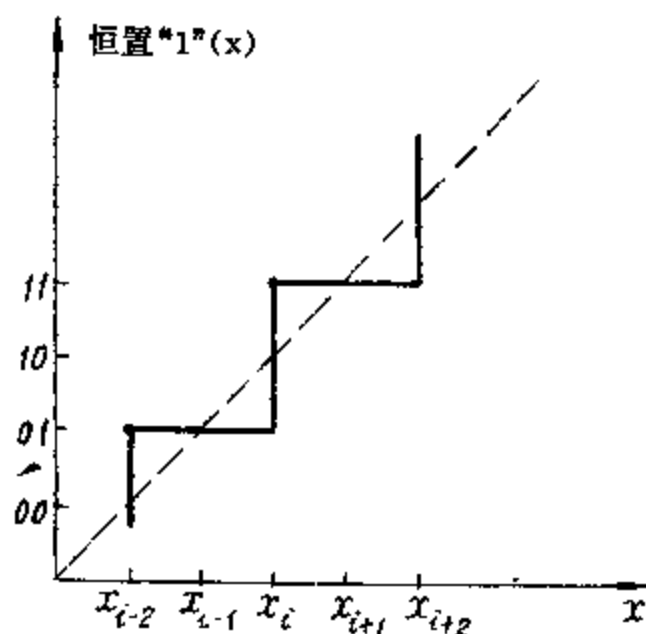
间以及 x_{i+3} 和 x_{i+4} 之间是按截断法形成结果, 其它各处是按舍入法形成结果。这样, 舍入法所形成的在理想曲线上的正误差被截断法所产生的负误差抵消, 使平均误差接近于零。



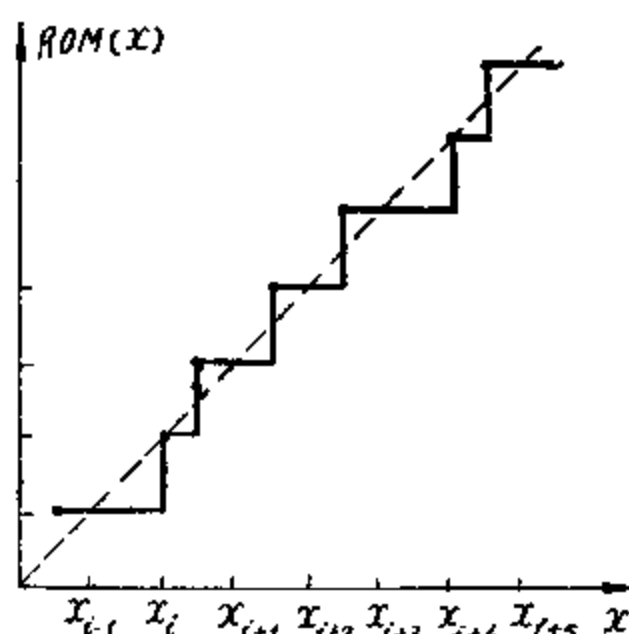
(a) $r_m = 2, m = 2$, 附加位位数 $d = 1$



(b) $r_m = 2, m = 2, d = 1$



(c)



(d)

图 2.4 各种下溢处理方法的误差

由上述可以看出, 截断法最省事, 速度最快, 但平均误差较大; 而 ROM 法则是即能比舍入法快, 又能调节平均误差, 使之接近于零, 但设备量增加了。目前机器多用舍入法或恒置“1”法。

1.3 自定义数据表示与向量数据表示

1.3-1 自定义数据表示

自定义数据表示指的是带标志符的数据表示和描述符。

一、带标志符的数据表示

在 § 1.1-1 中讲述了带标志符数据表示的由来。这种数据表示早在六十年代初期即已开始采用。例如, 当时 Burroughs B-5000 使用了一位标志位来区分操作数和描述符。它的后继者 B-6500 和 B-7500 则在每一个字中采用三位标志字段来区分八种类型。70 年代制成的

R-2 试验性计算机甚至采用 10 位标志符，其中两位用来指明是数值、控制信息、地址还是指令字。余下的八位中一位是奇偶位，一位是写封锁位，两位是软件定义的捕捉位，四位是直接标志位。

当指明该字是数值时，四位直接标志位的十六种不同组合可用来指明这个字是二进制数、十进制数、实数、复数、字符串或整数等等，还可以用来指明是单精度操作数还是双精度操作数。当指明该字是地址信息时，该四位直接标志位用来表示是绝对地址还是相对地址以及是否是链结中的地址等，为可变长数据和向量数据及阵列数据提供支持。当指明该字是控制信息时，该字段可以用来指明这个字是相对控制字还是绝对控制字。另外，还能指明该字是否未定义的。

看出，标志符虽然主要用于指明数据类型但还可用于指明机器内所用信息的各种类型。标志符对高级语言程序来说是透明的，它是由编译程序建立的。标志符数据表示的主要优点如下：

(1) 简化指令系统

前面讲过，在 Von Neumann 型机器结构中，是由指令指明操作数的类型，因此对同一种运算，需要为每种数据类型提供各自的指令。例如 IBM360/370 系统的加法指令就有九种。而在带有标志符机器的指令系统内，加法指令只需有一条，至于要执行的是哪种数据操作，则是由操作数的标志符来决定。由于标志符数据表示能提高指令的通用性（例如加法指令不必如一般机器那样分为十进制加法、二进制定点加法、浮点加法、逻辑加等等），所以也简化了程序设计。

(2) 便于实现一致性校验

采用标志符数据表示，能由机器硬件直接快速检测出很多种程序设计错误。例如，指令操作数是错误定义的（如乘法指令的操作数若是字符串则必然是错了）、不相容的（如试图把一个浮点数加到地址上去）或源操作数中有未被定义的值等等，从而简化和加快了程序的调试。而在一般机器中，这些校验是全由编译软件完成，这当然需要化费额外的处理机与存储器之间的数据传送以及额外的指令执行时间。

(3) 机器能自动实现数据变换

如果操作数是相容的，但是有不同的长度和表示，则机器能自动地把操作数变换成具有相同的数据表示，而后才进行运算。例如，可定义成机器加法指令的操作数中，当一个整数，另一个是浮点数时，不算出了程序设计错误，由机器硬件或固件把整数先变换成浮点数，而后再进行相加。由于经硬件或固件直接变换要比用软件变换快得多，从而可缩短解题时间。

(4) 简化编译程序

在一般机器中，目的代码的形成需要进行非常细致的语义分析。例如，当编译程序碰到“+”算子时，它必须检查算子要形成的是哪一种加法指令，而在带标志符的机器中，编译程序只需形成通用的加法指令即可。这当然会加快编译过程。

(5) 支持了数据库系统实现与数据类型的无关性

就是说，数据库系统可做到它的同一个程序能不加修改地适用于多种数据类型，例如，即能适用于十进制数据，又能适用于二进制数据。在一般 Von Neumann 型机器中要使程序具有这种与数据类型的无关性要求，其实现是很费事的，因为它的机器指令本身包含了操作

数类型和长度的信息。

另外，由于每个字都可以有“软件定义捕捉标志符”，就更便于实现程序跟踪，为软件调试提供更好的支持。

标志符表示的根本缺点在于每个字都因增加了标志符而使字长增长。直观来看，如果把4位标志符加到32位信息上，则存储器总位数和成本似乎需增加12.5%。但是进一步分析，情况并不一定如此。

首先，标志符数据表示能缩短指令操作码所需占用的总位数。举一个简单的例子来说明，设系统结构x具有150种不同类型的指令，从而需用8位操作码表示。设系统结构y，若采用带三位标志符的信息表示，则它的指令系统可简化成只需50多种指令，即操作码可缩短成6位。现在我们来分析这两种系统结构对存储容量的要求。设一个程序所需操作码位数和标志符位数的总和为B，而I为程序中机器指令的数目。那么对于x系统结构， $B_x = 8I$ 。如果对上述两种系统结构，每条指令需访问两个操作数，而且每个操作数在该程序中平均被

访问R次，那么对y系统结构， $B_y = 6I + 2 \times 3I/R = 6I \left(1 + \frac{1}{R}\right)$ 。二者之比值为：

$$\frac{B_y}{B_x} = \frac{6I \left(1 + \frac{1}{R}\right)}{8I} = 0.75 \left(1 + \frac{1}{R}\right)$$

显然，若此比值小于1，那么带标志符的y系统结构所需存储容量反而可以减小。

在程序的实际执行中，R是大于1的。当R=3， $B_y/B_x = 1$ ，这时二种系统结构化在操作码和标志符的总位数相等。76年有人指出，对一组程序的实测结果是R值约为10.4，如果这个值是有代表性的话，那么 $B_y/B_x = 0.82$ ，即带标志符系统结构对典型程序所用的操作码和标志符总存储位数只有不带标志符系统结构所需操作码总位数的82%。

这就是说，系统结构设计者不应只看到标志符所化的位数，而还应看到操作码位数的缩短，尤其还应看到一个操作数是会多次被访问。因此，一个操作数所带的标志符其实是被多次应用的，但操作码所多化的位数却是每执行一条指令都要反映出来，即它的冗余量可能更大。还要认识到，系统结构设计者应对程序设计语言和程序特征有比较深入的了解，才能设计好机器，对R值的分析就是一例。

其次，由于带标志符系统结构能够由硬件自动执行数据类型变换和诸如一致性检查等各种运行校验，因此还可缩短编译形成的目的程序的长度，即减少了目的程序所占用的存储空间。下面以具体例子来说明。例如，若一个PL/I程序定义变量I为定点二进制数，而变量A和D是定点十进制数。对于语句 $D = D + A$ ，由于操作数是相同的十进制数，IBM370的PL/I优化编译程序所形成的目的代码只需6个字节，但是对于 $D = D + I$ 语句，由于操作数不是同类型，其所形成的目的代码达64个字节，如表2.5所示（表上的执行时间是对于370/145机的）。这里，变换和校验用的程序是按排在目的代码之内，凡是用到的地方都有，因此在目的程序内是多处重复出现，即费存储单元，又增长目的程序的执行时间。而在带标志符的机器内，变换与校验是由机器硬件实现，即节省了存储单元，又加快了执行速度。这些功能的硬件实现，对于微程序机器，需在控存中增加相应的微程序。但是，在控存中它只需出现一次，而不是象驻留在主存的目的程序中那样是成百上千次重复（若用机器子程序技术，虽然所用存储单元可减少，但执行时间会过长）。

表 2.5 PL/I 语句在 IBM370 上形成的目的代码长度 (字节数)

PL/I	目 的 代 码		
语 句	机器指令条数	长度 (字节数)	执行时间 (μ s)
$D = D + A$	1	6	13.1
$D = D + I$	13	64	407.6

综上所述,带标志符数据表示的优点是明显的。虽然每个数据字的长度会增加,但从整个程序看,所需存贮容量并不会增加多少;加上存贮器每位价格的持续下降,这种能缩小高级语言与机器语言间的语义差别的标志符技术已日益为人们所接受,并正研究如何进一步扩大它的用途。

当然,不一定每个字都得有标志符。例如在阵列中,就不需要每个元素都加标志,因为在一般情况下,阵列中的每个元素具有相同的属性,可以给整个阵列只配一个标志,这就是数据描述符。

二、数据描述符

数据描述符不是用于描述简单的基本数据,而是用于描述复杂的和多维的数据,如阵列和记录等。标志符是和每个数据相连,合存在一个存贮单元内,而描述符则和数据分存,机器经描述符形成访问每个元素的地址。

下面以 B-6700 的描述符为例来说明,它的组成如图 2.4 所示。前三位为“000”表示该字为数据,若为“101”表示该字为描述符。若所描述的是数据块,“长度”字段指明块内的元素个数,“地址”字段指明首元素的地址。

描述符

1 0 1	各 种 标 志 位	长 度	地 址
-------	-----------	-----	-----

数 据

0 0 0	数 值
-------	-----

图 2.4 B-6700 数据描述符格式

经描述符取操作数的过程如图 2.5 所示。按指令操作数地址 x, y 访存,若取来的字,其前三位为“000”,则它就是所需的操作数;若为“101”,表明它是描述符,将它取到描述符寄存器,由它的“长度”和“地址”字段经地址形成逻辑形成操作数的地址,再访存取操作数。当然,对于数据块,访存取到寄存器的描述符,能应用于块内的所有元素,不必每次访存取元素时都去加上访存取描述符的操作。这样,只需由这样一条指令就能执行对整个数据块的运算。

B-6700 还可把描述符按树形联结以描述多维数据结构。图 2.6 表示如何用描述符描述

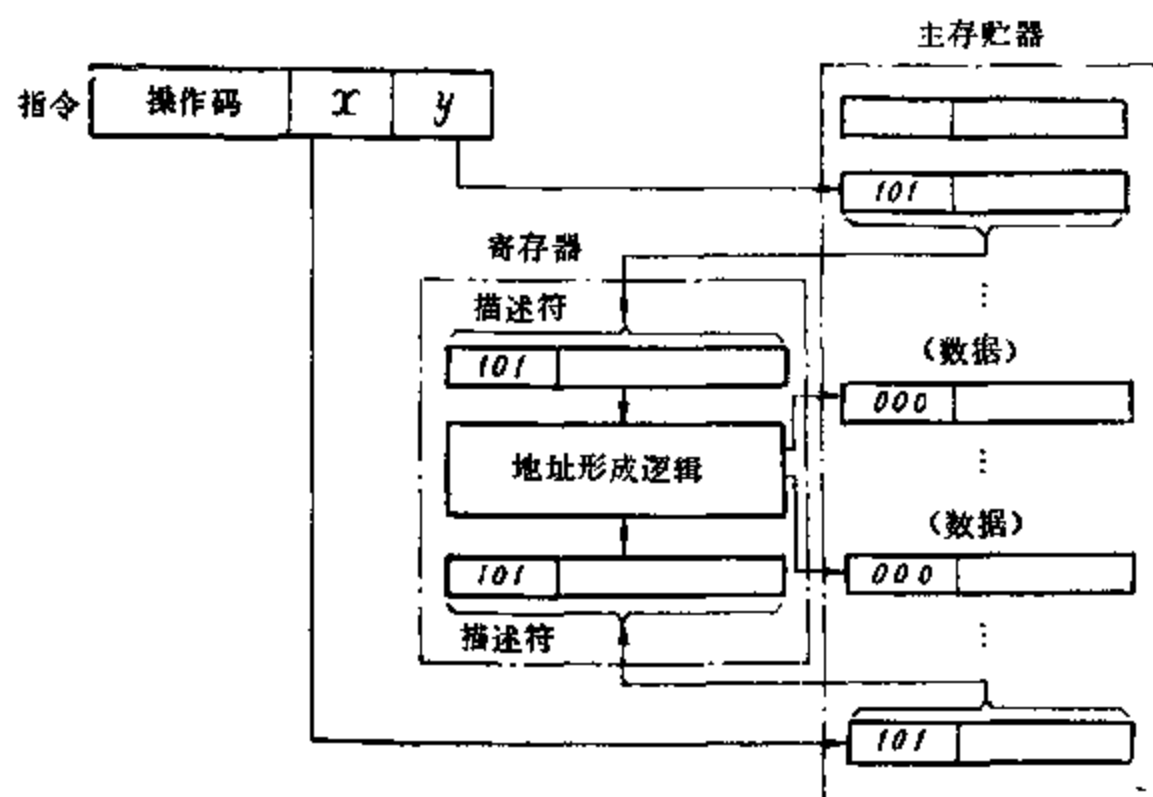


图 2.5 经描述符访存取操作数

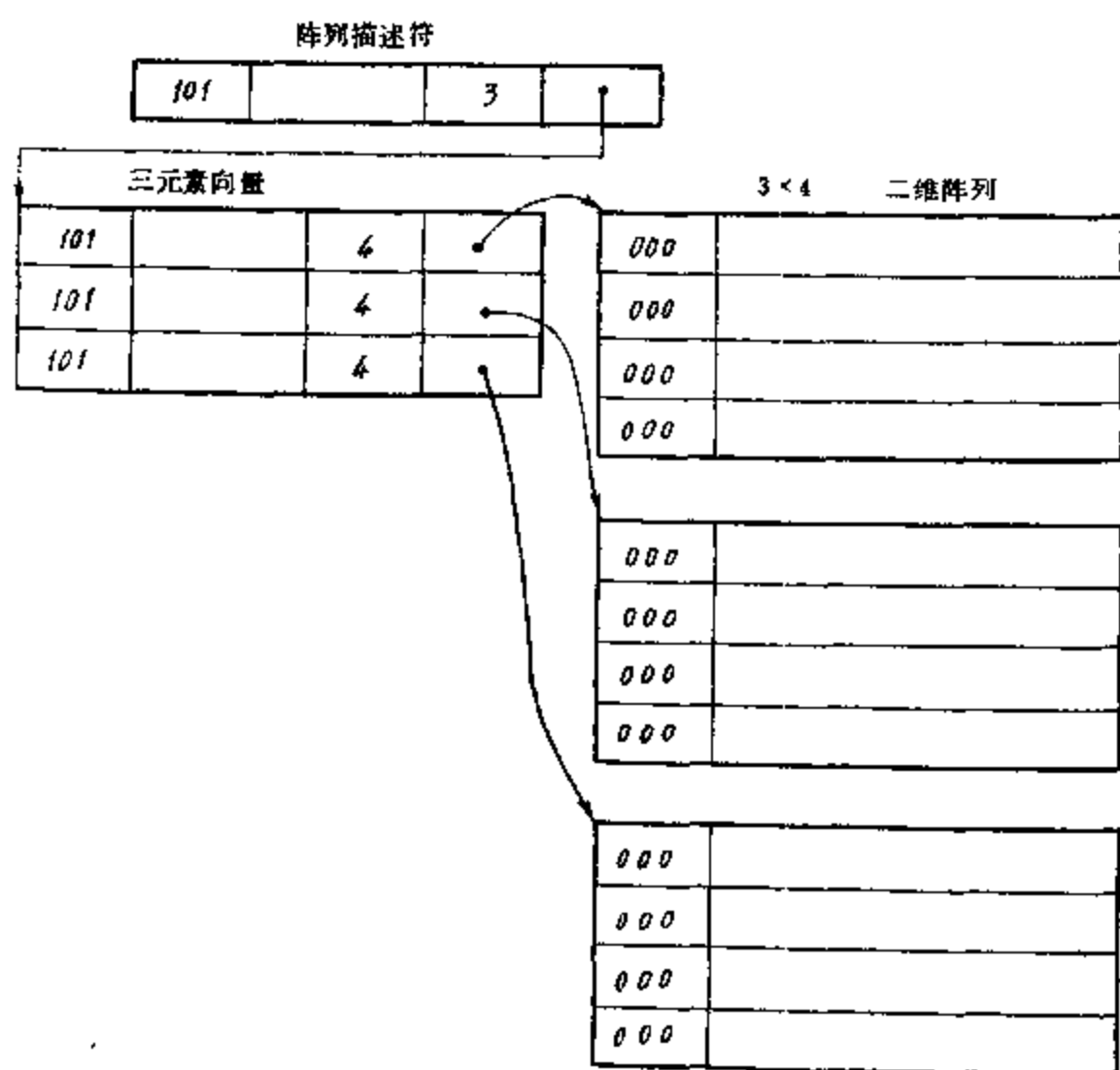


图 2.6 用描述符描述二维阵列

3×4二维阵列。阵列描述符指向三元组描述符向量，而每个描述符元素指向相应的四元组向量。

看出，用描述符方法实现阵列的索引要比用变址方法实现方便得多，而且也便于检查出程序设计中的阵列越界错误。

用描述符方法能实现用一条指令控制整个阵列的运算（如阵列相加、相乘），这对简化编译中的代码生成是有好处的；它还可以比变址法更快地形成元素地址，从而能加快阵列运

算。然而，这些和阵列、向量各元素能否同时、并行运算并无关系。这就是说，采用描述符方法，对阵列、向量运算，各元素即可能是同时、并行运算，也可能是同时只执行一个元素的运算，这取决于运算器和主控制器的设计。

1.3-2 向量数据表示

前面讲过，向量是在解很多题目中广泛使用的一种数据结构，诸如多项式求解、气象数据处理和核物理数据处理等都广泛运用向量运算。如何为向量数据结构的实现和快速运算提供更好的硬件支持是系统结构设计者在七十年代很注意的一个方面。不只是可用描述符描述向量，就是不采用自定义数据表示的机器，有的也已增加了向量数据表示，如 STAR—100 和 CRAY—1 等就是如此。

对于有向量数据表示的机器，对向量的运算需经过二步，第一步是做准备工作，即把源向量的长度和起始地址置定好，一般是从主存把这些参量取来送到寄存器；第二步就是用一条向量指令直接对整个向量的所有元素进行运算。这当然比没有向量数据表示，而是借助于变址操作实现向量运算的机器要方便得多、快得多。因为对后者，变址（索引）值的置定，按变址值形成元素地址，改变变址值，将改变了的变址值与最终值进行比较，按比较结果进行转移等，都得由各自的机器指令实现。看出，不论向量多长，对于有向量数据表示的机器，第一步准备工作所需时间是一样的，因此，若向量过短，则第二步的实际运算时间反倒可能比准备工作时间短，这当然就不合适了。

对于如下

```
DO 40 I=10,1000
  40 C(I)=A(I+5)+B(I)
```

这种 DO 语句，具有向量数据表示的机器可以用以下这样一条向量相加指令

$$C(10:1000) = A(10 + 5:1000 + 5) + B(10:1000)$$

实现。显然，对参加运算的每个向量都需指明其基地址、位移量和向量长度，见图 2.7。基地址指向该向量的第一个元素，起始地址指向实际参加运算的第一个元素，图中每格存一个元素。向量长度用于校验所形成的元素地址是否越界。设置位移量及能由它和基地址形成起始地址便于实现斜排 (Skew) 向量运算。如果一条机器指令的三个向量的所有这些参数都由机器指令的相应字段直接指明，那机器指令势必过长，而且对于编译及程序的使用都不方便，

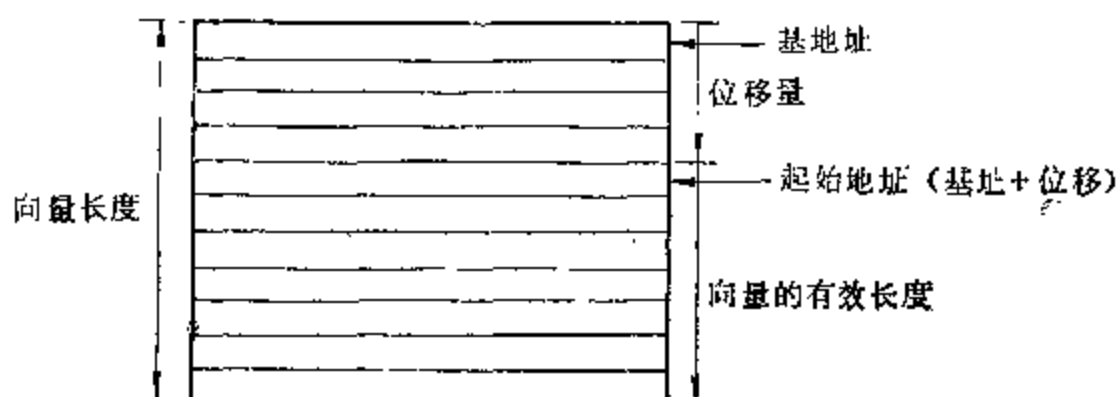


图 2.7 向量的编址

所以 STAR—100 在机器指令中是指明存这些参数的寄存器号，再由寄存器的内容来确定向量参数（基地址、位移量、向量长度）。

STAR—100 还能够实现对稀疏向量的压缩以节省存稀疏向量所需的存贮单元，如图 2.8(a)所示。由排序(掩蔽)位向量指明哪些元素是被压缩了的，排序位向量在主存中占一个存贮单元。机器有稀疏向量运算指令，能直接由硬件控制稀疏向量的处理，图2.8(b)就是A、B稀疏向量的相加。当然，A、B稀疏向量的排序位向量需事先取到相应的寄存器。稀疏向量在线性规划、建筑工程计算、运输分析和航空工程中用得很多，为它提供硬件支持是需要的。

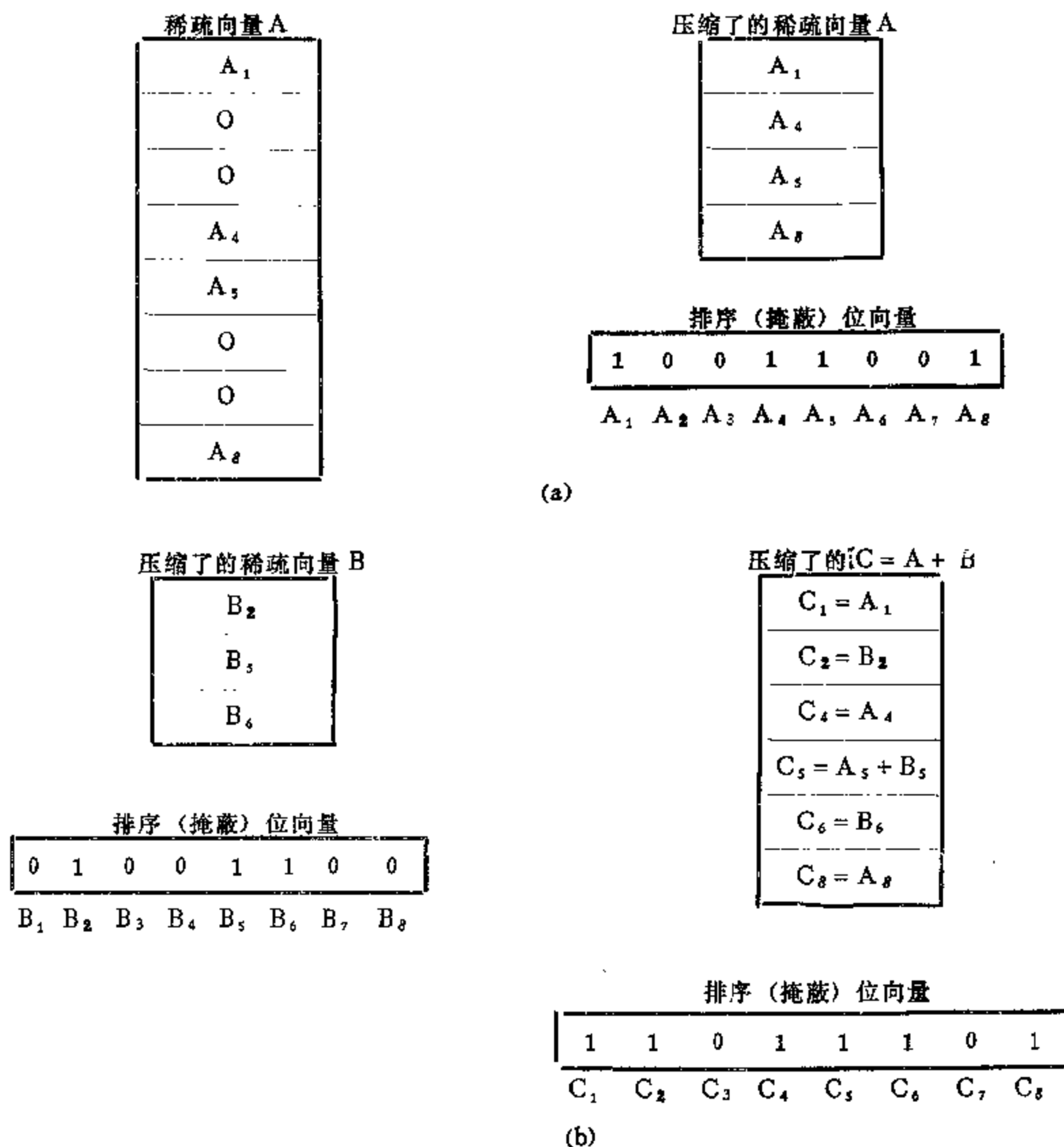


图 2.8
(a) 稀疏向量的压缩 (b) 稀疏向量的相加

当然，引入向量数据表示不只是为了便于快速地形成元素地址，而更主要的是便于实现把整个向量的元素(即整块操作数)或是把整个向量分成几块，成块地予取到中央处理机以加快对向量的运算。又由于向量运算是对所有元素执行相同的运算，这很有利于发挥流水技术的效能，使机器速度能得到显著的提高。这些在第三章叙述流水技术时结合 CRAY-1 机器再讲。

综上所述，数据结构的发展当然是领先于机器的数据表示，随着器件价格的进一步下

降,系统结构设计者完全应该也有可能为数据结构的实现提供更好的支持,除了上面讲的以外,基于相联存贮技术的相联型数据表示是又一个例子。当然,每种新的数据表示的引入往往带有某种冒险性,一是前面讲过的,它的通用性和利用率究竟怎样,在机器制成之前是难以取得明确的结论的;二是程序设计者对于如何使用和发挥这种数据表示的功能需要有比较长时间的摸索和适应过程。因此,新的数据表示一般是首先使用于巨型机和专用机;而后,随着使用经验的积累,不断改进,再引用于通用系列机中。数据表示的这种“下移”在八十年代会进一步加速。

§ 2 地址形成

上一节讲了机器的数据表示。数据需装进主存后,才能被机器指令所访问,这可以有按地址访问、按堆栈访问和按内容访问等方式。按内容访问需用到相联存贮器片子,这种方式在第五章再讲。按堆栈访问在取操作数时,不需由指令地址码指明操作数的地址,而是只能从所谓栈顶取得,这种方式在本章的§ 4再详述。按地址访问用得最普遍。它的主存是由按地址访问的随机存贮器片子构成,需由机器指令指明或形成操作数所在单元的物理地址之后才能访问。对直接编址、间接编址、相对编址和变址编址等编址方式大家是熟悉的,因此本节着重于讲述如何由机器指令地址码形成操作数有效地址,以及为什么要如此形成。

我们先讲述逻辑地址空间和物理地址空间的基本概念,而后以 DJS—200 系列机为例讲述有效(物理)地址的形成。

2.1 逻辑地址空间与物理地址空间

对最初的计算机,程序员编制的用户程序(包括数据)在主存中的存放位置是由程序员决定和指明的。如果把由程序员编制的程序地址称为逻辑地址,而把程序在主存内的实际地址称为主存实地址,那么,对最初的计算机,逻辑地址和主存实地址是相同和一致的。那时,由逻辑地址构成的逻辑地址空间(程序空间)和由主存实地址构成的物理地址空间(实存空间)是相同和一致的。

后来,随着汇编程序、编译程序和操作系统的出现,主存中至少是存了二道程序,即系统程序和一个用户程序。此时,源程序不是用机器指令代码和指令、数据在主存的地址编址,而是以运算符、符号、标量和数据说明等编写。由这些符号构成的空间称为符号名称空间或名称空间。机器通过编译程序对源程序进行处理。然而,并不是把名称空间直接翻译成物理地址空间。

主存只有一个从零地址开始的编址空间,但是程序有多道。如果各道机器语言程序的编写都是从零地址开始编址,那必然不能和主存编址相符;但各道程序又不能不从零地址开始编址,因为在编制程序时,程序员不可能予知他所编制的这道理程序会存在主存中从哪个实地址开始的空间,以便从这个实地址开始编址。这样,逻辑地址空间与物理地址空间必然是不相符的,如图 2.9 所示。图中, A 道程序的逻辑地址空间是从 0 地址到 m 地址,但它在主存的物理地址空间却是从 a 地址到 a + m 地址。因此,就需由装入程序在把 A 道程序装入主存时,进行逻辑地址空间到物理地址空间的变换。

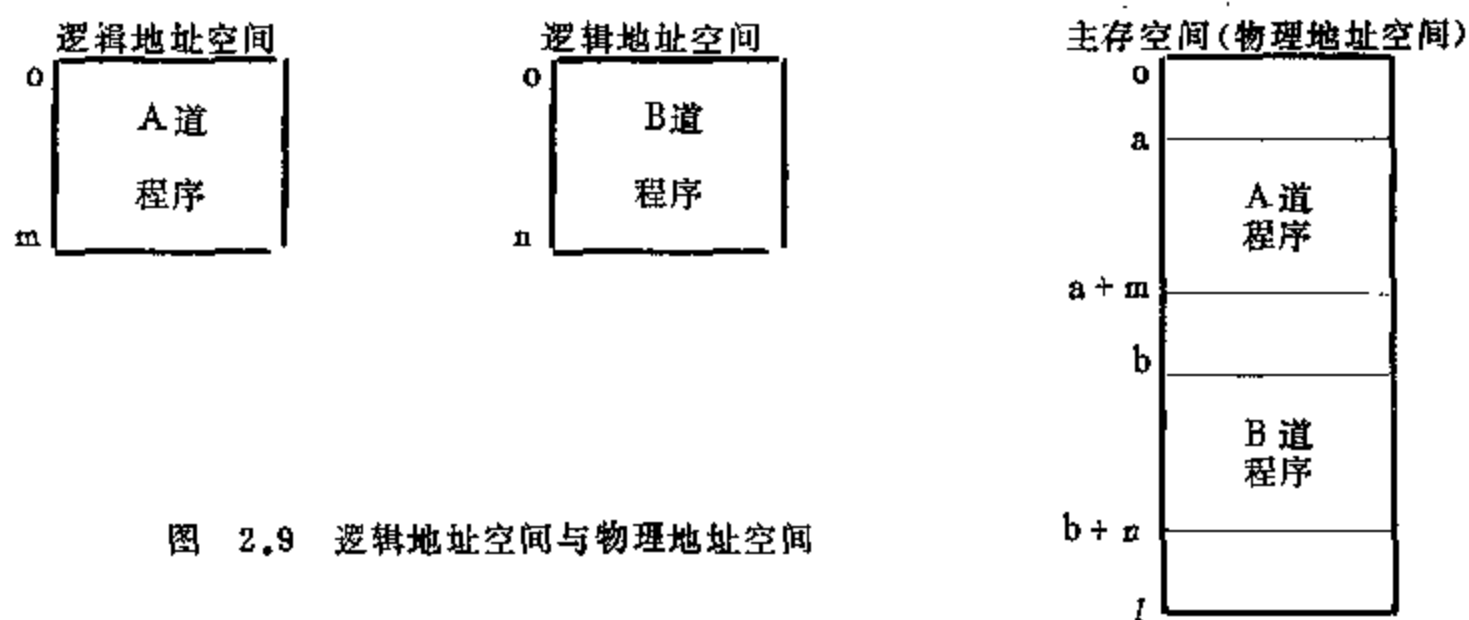


图 2.9 逻辑地址空间与物理地址空间

这种变换必然要包括改变 A 道程序的指令地址码，如图 2.10 所示。显然，对于操作数以及转向地址中的直接编址、间接编址以及变址编址都需要改变，即都要加上 a 值。这样，装入程序既要控制把 A 道程序装入主存，又要在装入时把相应的指令地址码修改好。然而，并不是所有指令的地址码都需要进行修改，例如，对直接操作数和相对地址（相对于该条指令所在的地址），就不能加上 a 值。因此，装入程序在修改地址码时，还得要能区分是什么样的地址码。

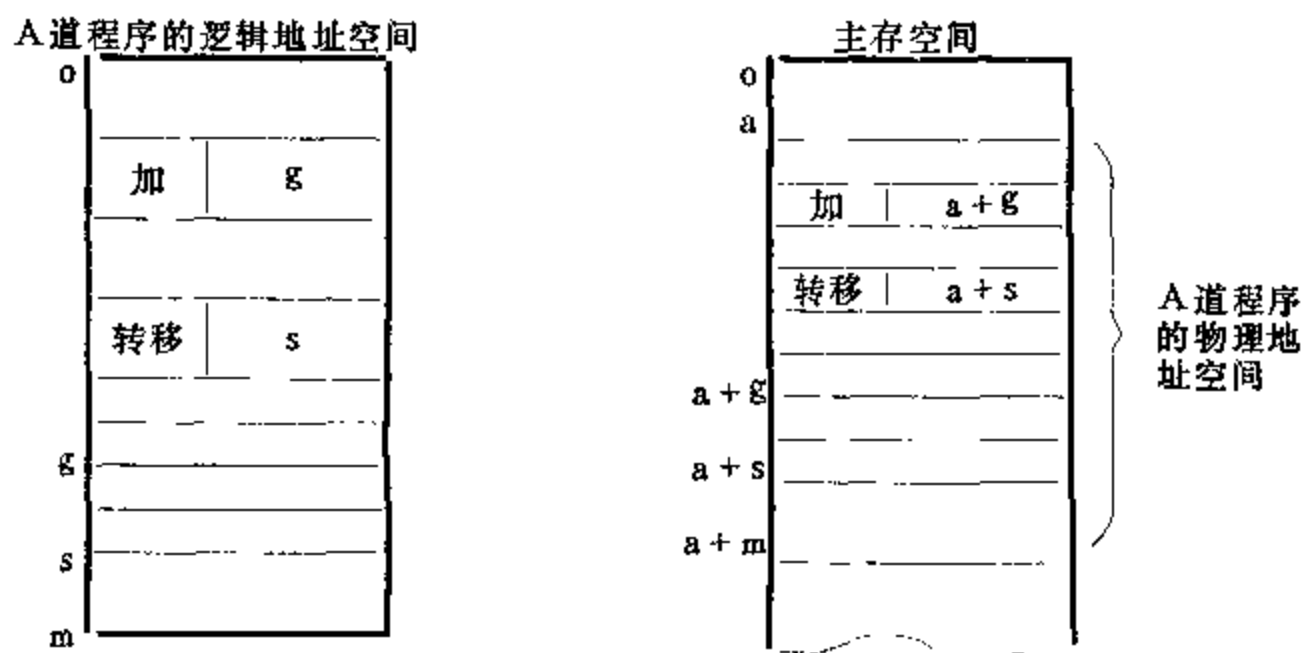


图 2.10 逻辑地址空间到物理地址空间的变换

这样，由编译程序对名称空间的翻译，就不能要求直接形成物理地址空间，而是只把该道程序的名称空间翻译成逻辑地址空间（即编译的输出是从零地址开始编址的目的程序），而后再由装入程序把目的程序的逻辑地址空间变换成目的程序的物理地址空间，执行程序的定位。

上面说过，变换要改变指令的地址码，这是应用了 Von Neumann 型机器指令可修改的特点。采用这种方法，应在目的程序执行之前把地址空间变换好，而且在整个程序的执行过程中，物理地址空间的位置是不能动态再改变的。诚然，原始 Von Neumann 型机器的指令可修改特征对早期计算机起过很好的作用。大家知道，通过程序的循环和修改指令的地址码，就能用一段短程序对大量数据执行重复的相同运算。

但是，到了六十年代，随着多道程序的广泛应用，一段程序（如常用子程序）可能会被多个用户程序所公用。这时，修改指令地址码的方法就是危险和不方便的了，例如，当某个

用户程序在调用这段程序进行修改地址码时,若不当地改错了,则不仅是要影响到本道程序,还会使其它道程序也出错。而且,指令(包括地址码)允许修改这种做法对于诊断和程序调试都是不利的,因为指令既然可改变,一旦出错就比较难于找到故障点。还有,指令的允许修改还会给重迭、流水技术的采用带来困难(在下一章再讲述)。这样,从六十年代以后,不准修改指令的做法日益被人们所肯定,这当然给系统结构设计者提出了新的要求。

那么,在不许修改指令地址码的前提下,如何实现逻辑地址空间到物理地址空间的变换呢?由图 2.10 看出,就装入程序所执行的地址空间变换来看,除了把该道程序装入主存以外,还得给程序的相应地址码加上该道程序在主存中的起点地址(即 a)。既然不准修改地址码,那这个相加操作就只能经硬件实现。吸收变址编址的思路,就出现了基址编址的方法,如图 2.11 所示。现在,不能以指令地址码(逻辑地址)直接去访主存,而需加上存在

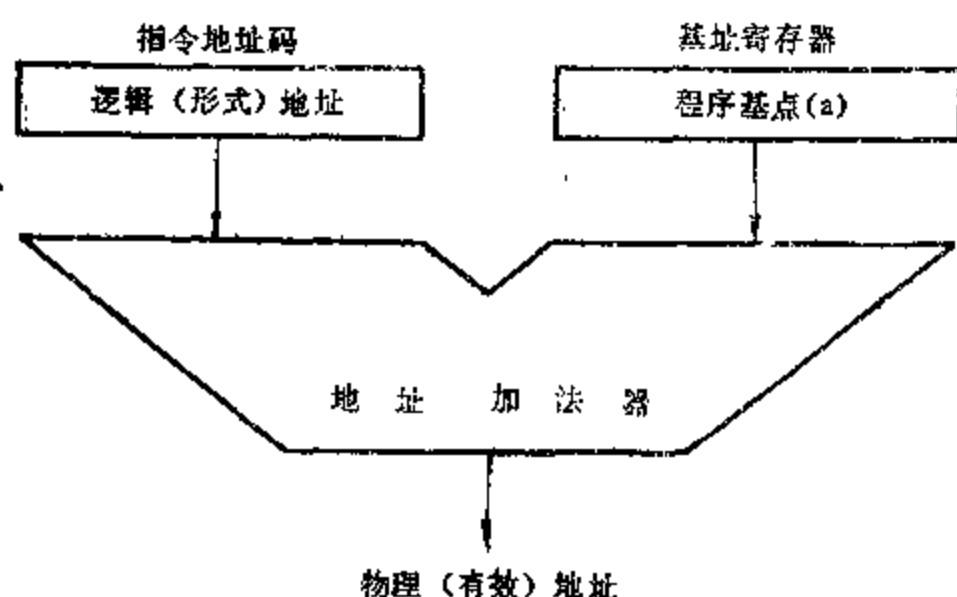


图 2.11 基址编址

基址寄存器中的程序基点地址(简称基址),形成物理(有效)地址后才能去访存。

上面也讲过,在变换地址空间时,并不是所有的指令地址码都要修改。因此,在指令中还得有相应的标志,指明指令地址码是否需加基址值。变址编址和基址编址当然不是同一概念,前者是对诸如向量等数据块的支持,而后者则是对逻辑地址空间到物理地址空间的变换提供支持。因此

此二者是会并用的,IBM 370 地址码格式中的一种就是这样,其格式如下:

操作码	R_1	X_2	B_2	D_2
-----	-------	-------	-------	-------

机器有 16 个通用寄存器,可以存放操作数、变址值和基址值。第一个操作数取自通用寄存器,其寄存器号由 R_1 指明;第二个操作数取自主存, X_2 为变址值所在的寄存器号, B_2 为基址值所在的寄存器号; $X_2 = 0$, $B_2 = 0$, 分别表示不加变址值和不加基址值。这样,有效地址经地址加法器如下形成*:

$$\text{有效地址} = (X_2) \cdot (X_2 \neq 0) + (B_2) \cdot (B_2 \neq 0) + D_2$$

有了这种基址编址的支持,装入程序的任务就可简单得多,它只需控制把程序装入主存,并把该道程序在主存实际位置的基址值送入相应基址寄存器就可以了。而后,在程序每条指令的执行过程中,再经加基址值把逻辑地址变换到物理地址。这种在执行每条指令过程中才形成访存物理地址的方法称为动态再定位,而前述那种先把整个程序的地址修改完,才执行程序的方法称为静态再定位。

综上所述,基址编址实质上是把原来经软件实现(如经加法指令执行相加)的物理地址形成,改进为经地址加法器硬件形成,这当然使速度得到提高。

* IBM 370 指令中定义 x 为变址字段, B 为基址字段, DJS-200 系列机指令系统中,由于使用拼音,定义 B 为变址字段,请读者看书时注意区分。

然而，对这种不是直接取自指令地址码，而是经主控制器硬件形成的地址，就需判定其有效性，其内容可以是多方面的。例如，对某个用户程序，要判定所形成的有效地址是否越出分配给该用户的物理地址空间，以防止侵犯系统软件或其它道程序。参看图 2.10，分配给 A 道程序的是从 a 到 $a+m$ 的物理地址空间，称此空间的始点（即基址值 a ）为下界，而终点（即 $a+m$ ）为上界。这样，如果

有效地址 $>$ 上界 或 有效地址 $<$ 下界

即出现越界错，这时就需经中断转入越界分析和处理。当装入程序把 A 道程序装入主存时，不仅要置定好下界值，而且要把上界值置入相应的上界寄存器，并在形成每个有效地址时，都要判越界否。这就是所谓存贮保护，即由硬件（辅以软件）保证每道程序只能访问到给定的存贮区域。上述这种存贮保护称为界限（界地址）保护。存贮保护还包括访问方式保护，例如，对于多个用户公用的程序，就要有写保护（即只能读出，不能写入），以防止某个用户的程序可能破坏这个公用程序。访问方式的其它保护及其它存贮保护方式在第五章讲。

这种不是直接用指令地址码去访主存，而是根据各种需要，经硬件把它变换成有效地址后再去访存的思路是很有意义的，这是系统结构设计者应注意研究的一个方面。

六十年代初、中期系统结构就已提供的这种上、下界寄存器技术和相应的存贮保护等，是对操作系统存贮管理诸如动态再定位等的重要支持，是地址映象技术（由逻辑地址映象到物理地址）的重要进展。这些，显著地改善了多道程序的管理和实现。但是，加下界的这种地址映象方法，程序员的编址空间是小于，至多等于主存容量。它的进一步发展是六十年代末提出，七十年代初开始应用的虚拟存贮系统，它使每个用户程序设计者能具有比主存容量大，等于虚存容量的编址空间，这是地址映象技术具有深远影响的重大进展。关于它，在第五章将详细讲述。

下面附带讲讲物理地址空间的信息分布问题。前面讲过，可以有按位编址、按字节编址和按字编址等各种方式；而且，在同一个机器内，可能同时采用这些信息编址。那么，在同一个主存空间内，各种编址的信息应该怎么存贮和混存呢？下面结合 IBM370 进行分析。IBM370 的信息有字节（8 位）、半字（双字节）、单字（四字节）和双字（八字节）等宽度。设主存按字节编址，宽度为 64 位，即一个存贮周期可读出 8 个字节。若各种宽度的信息可以任意存贮，就可能出现如图 2.12(a) 那种情况，那就是在

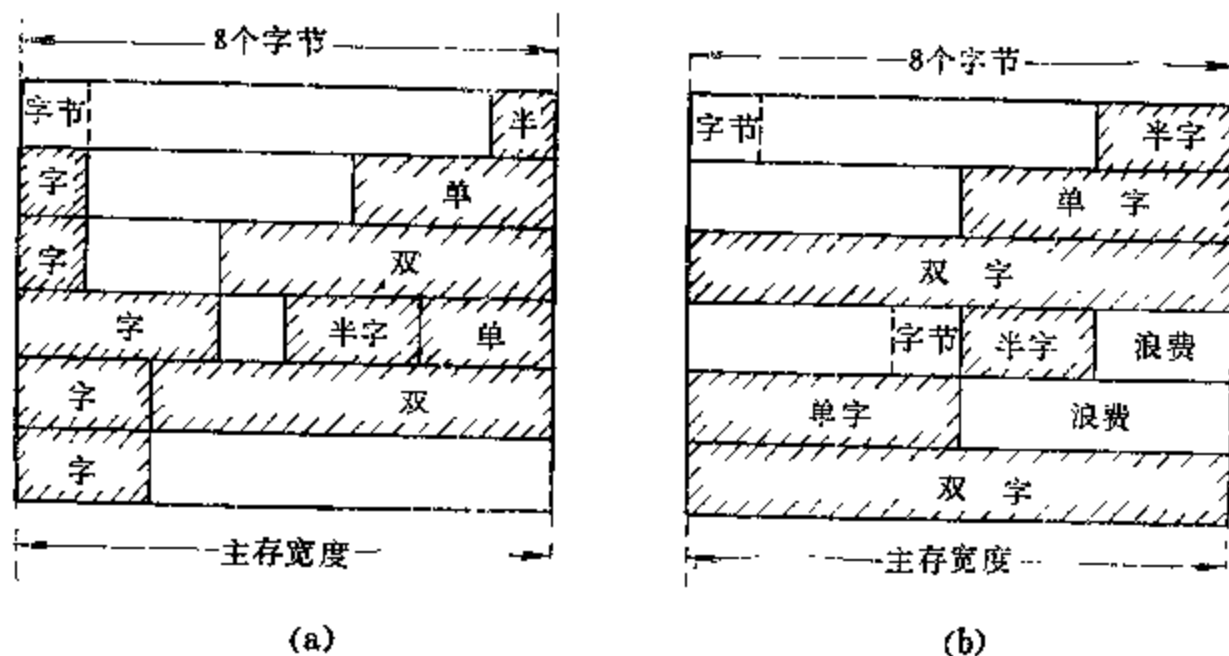


图 2.12 各种编址信息的存贮

信息宽度 \leq 主存宽度

时, 也需化二个存贮周期才能访问到; 但若限制成只能如图 2.12(b) 那样存贮, 则在信息宽度 (最长为双字, 8 个字节) 小于或等于主存宽度 (8 个字节) 时, 总是只需化一个主存周期就能访问到。若各种宽度的信息的地址是以该信息的首字节的字节地址来表示, 为要防止出现图 2.12(a) 的情况, 就必需要求从该信息的首字节起, 到主存边界之前, 肯定要能存得下该信息的所有字节。显然, 这要求各种信息的地址必需是:

字节地址 $\times \times \times \times$
半字地址 $\times \times \times 0$
单字地址 $\times \times 0 0$
双字地址 $\times 0 0 0$

因为只有这样, 当信息宽度 \leq 主存宽度时, 才不会出现如图 2.12(a) 那样一个信息跨主存边界存贮的情况。这就是所谓信息的整数边界, 即各种信息的地址 (用该信息的首字节地址表示) 必须是该信息宽度 (字节数) 的整数倍, 否则是地址错; 例如, 单字宽度为 4 个字节, 则单字的地址必须是 4 的倍数, 即其最末两位必须是零。信息的整数边界对于保证访问速度当然是必要的, 然而, 它却会造成存贮空间的浪费。例如, 若要顺序存贮的三个信息为半字、单字、双字长, 则如图 2.12(b) 下半部所示, 会出现六个空白字节的浪费。但若各种信息可以相连任意存贮, 如图 2.12(a) 下半部所示, 那当然不会出现空白字节浪费, 但有些信息 (如图中的单、双字) 的访问却需要化费二个存贮周期。

对于按位编址的机器, 如 STAR-100, 信息的存贮也是按整数边界, 但信息的地址值需是其位数的整数倍。

2.2 有效地址形成举例

下面结合 DJS-200 系列机的实例, 具体地讲述基址编址的有效地址形成。

DJS-200 具有 16 个通用寄存器, 用于存放操作数和变址值等; 但它的基址值不是如 IBM360/370 那样是存在通用寄存器, 而是存在专用的下界寄存器 (X_d) 中。它和 IBM360/370 不一样, 在指令中没有基址字段 (基址寄存器号)。它是三地址机器, 基本的指令格式如下:

操作码	L_1	L_2	B_2	d_2
-----	-------	-------	-------	-------

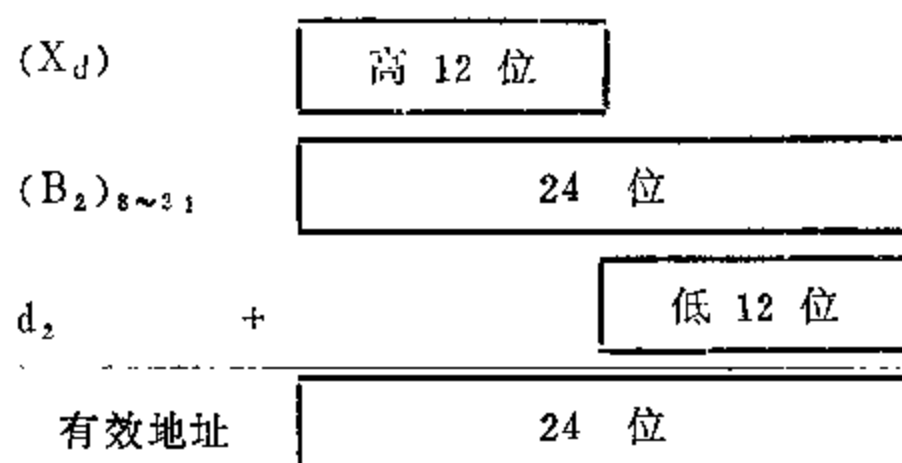
L_1 、 L_2 指明存第一操作数和运算结果的通用寄存器号, B_2 为第二操作数变址值所在的寄存器号。第二操作数可取自主存, 其有效地址如下形成:

$$(X_d) + (B_2) \cdot (B_2 \neq 0) + d_2$$

DJS-200 的地址码为 24 位, 单字长为 32 位, 通用寄存器为单字宽。为使有效地址可经一般双输入端加法器形成, 取变址值为 24 位, 对应 (B_2) 的第 8 位到第 31 位; (X_d), d_2 分别对应高、低 12 位, 如下图所示:

这样, 只需把 (X_d), d_2 合送入加法器的一个输入端, (B_2)_{8~31} 送入加法器的另一个输入端就可形成有效地址。当然, 对所形成的有效地址需进行判越界。

前面讲过, 若某段程序 (数据) 是主存内多个程序都要用到的, 那当然不应是各道程序



的物理地址空间内都重复装有相同的这段程序，而应是在主存内只有一套，且各道程序都能

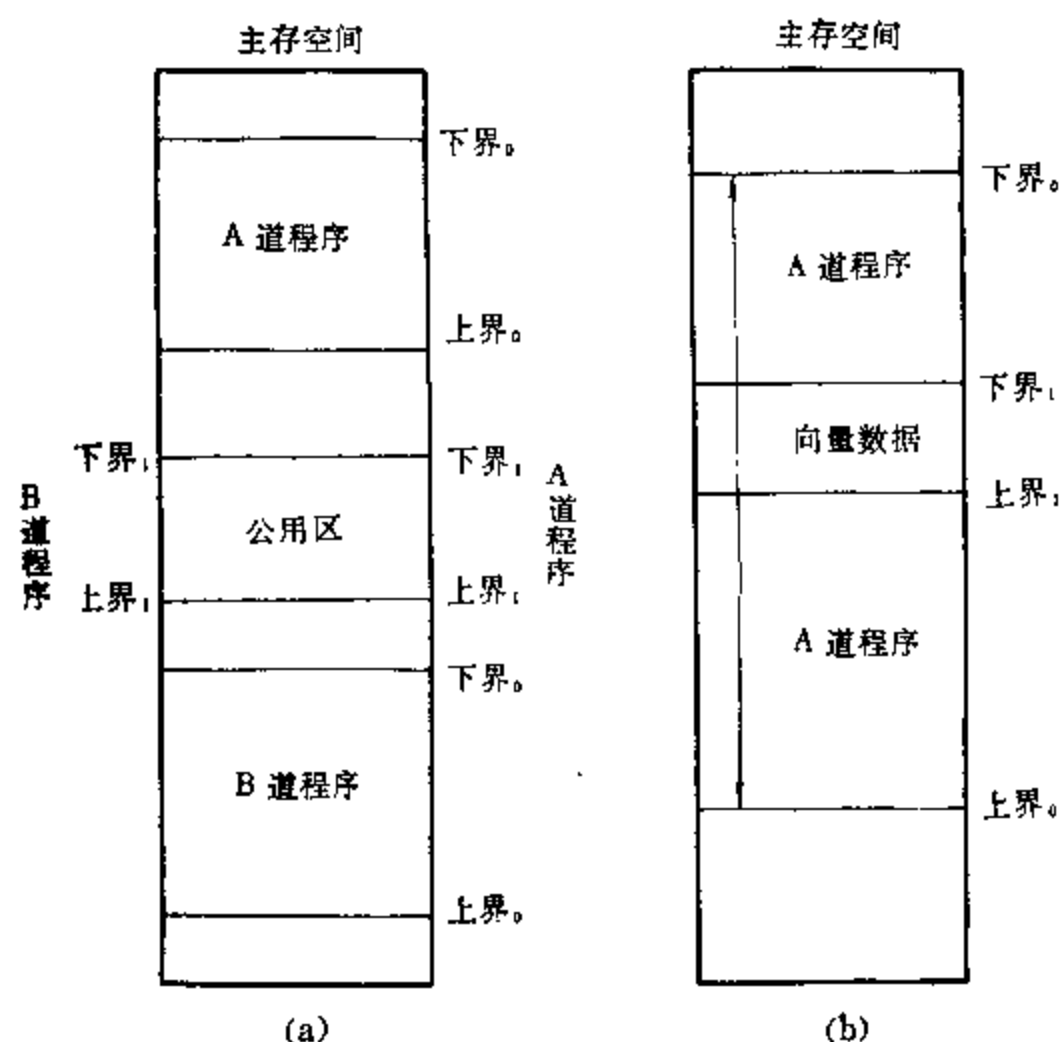


图 2.13 二对界的各种用法

访问到。可是，对于上述每道程序只占据一个连续空间（或称具有一对上、下界）的情况，显然做不到这点。为此，DJS-200 设置二对界，称为 0 界和 1 界，即机器内有二个下界寄存器和二个上界寄存器。这样，现行的每道程序就可有二个连续的空间，如图 2.13 所示。

二对界的用法是灵活的，例如，既可以如图 2.13(a) 那样用一对界解决公用区的使用问题；也可如图 2.13(b) 那样用于判定经变址形成的向量数据元素地址的有效性。由于变址法也是一种不直接用指令地址码，而经机器硬件形成地址的方法，若能判定经硬件形成的地址的有效性，这

对提高可靠性是有好处的。

既然每道程序可用到二对界，就必须有“界号指示位”，以指明所要访问的操作数是在哪对界内。由于在某对界内的指令，其操作数既可能取自指令所在的这对界内，如图 2.13(a) 中的 0 界内；也可能取自另一对界内，如 1 界内。但 DJS-200 的指令码中并没有“界号指示位”字段，因此它只得采用一些技巧性办法，用变址寄存器的第二位 $(B_2)_2$ ，或程序状态字 PSW 的第 34 位 PSW_{34} ，作为界号指示位。若 $B_2 \neq 0$ ，则以 $(B_2)_2$ 作为界号指示位。界号指示位为“0”表示由 0 界内取数，为“1”表示由 1 界内取数。这样，对于有变址操作的，通过给 $(B_2)_2$ 置定不同值，就可控制各个操作数是由哪对界取得；但对于不是变址编址的，则只能由 PSW_{34} 指明的那一个界取。由于 PSW_{34} 是对应程序所在的界，因此无变址操作的操作数只能由指令所在的那个界内取。应该说，这种加界安排方法是不符合系统结构的规整性（以后要讲到）要求，这是在指令码安排不下界号指示位的情况下所采用的不得已办法，它对机器的便于使用和程序的便于调试不会带来什么好处。

至于前面讲过的“写保护”，DJS-200采用给下界寄存器配“访问方式位”解决，若访问方式位为“0”，则在这一对界之内的区域可进行任何型式的访问，但若为“1”，则称此区域为保护区，只能读，不能写，若出现写保护区则发中断信号。

DJS-200配备了半固定存贮器（简称半固），所以数据亦可取自半固。由于半固只能读出不能写入，它的内容是破坏不了的，因此对它的访问没有必要采用界限保护和写保护等措施，但需有半固指示位。当其为“1”时，表示要访问的是半固，肯定不加界；当其为“0”时，表示要访问的是随机存贮器主存，则按上述规定进行加界。由于访半固不加界，当然无所谓越界检查。但是访问地址不能超过其实际的容量，否则认为是地址错。

§3 指令系统

指令系统是机器所具有的各种指令的集合，或称指令集。指令系统表征着计算机的基本功能，它是从程序设计者看的机器的主要属性和软、硬件的主要交界面。虽然指令系统的设计一直是系统结构设计者十分注意的一个重要方面；然而，应该说这一、二十年来进展是缓慢的，这点我们在下面要详细分析。

指令系统的设计主要包括操作类型、内容的设计和指令格式的设计。在本章的前两节讲了数据表示的选择是确定运算型指令的前提，讲了地址的形成对地址码格式的影响；本节在这些基础上讲述“指令格式的优化”和“指令系统的分析”。

3.1 指令格式的优化

指令字主要由操作码与地址码组成以及指令有零地址（堆栈操作）、一地址、二地址、三地址和四地址等方式，这些大家是熟知的。至于直接编址、间接编址、相对编址、变址编址和基址编址等方式在指令格式中如何表示，有操作码指明和寻址方式位二种方法。前一种方法是由操作码指明操作数的编址方式，例如，DJS-200的间接地址型（J型）和上一节讲过的基本型（LB型），其指令格式都是：

操作码	L_1	L_2	B_2	d_2
-----	-------	-------	-------	-------

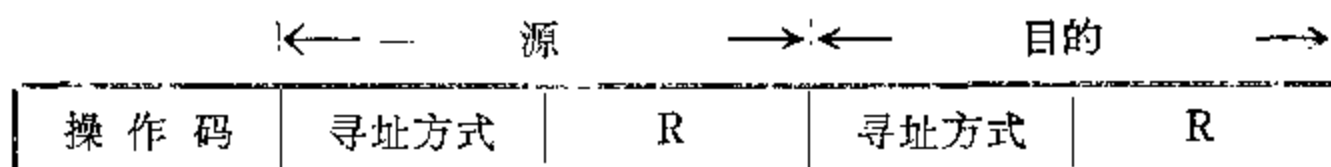
当8位操作码是 $01\times\times\times\times\times\times$ 为LB型，其第二操作数的有效地址是 $D_2 = (X_d) + (B_2) \cdot (B_2 \neq 0) + d_2$ ；但当8位操作码是 $11\times\times\times\times\times\times$ 为J型，其第二操作数则是间接编址，其有效地址是

$$((\dots((D_2))\dots))$$

DJS-200单字长为32位，地址码为24位，间接寻址时，取 $(D_2)_{8\sim 31}$ 为地址码，以 $(D_2)_0$ 表示间接位，若 $(D_2)_0 = 1$ ，则继续间接；若 $(D_2)_0 = 0$ ，则以此时的 $(D_2)_{8\sim 31}$ 为地址码访存取来的就是第二操作数。除寻址方式的这种差别之外，操作码为 $01\times\times\times\times\times\times$ 的指令与操作码为 $11\times\times\times\times\times\times$ 的指令，它们的操作内容完全一样。就是说，LB型的最多可达64种的指令，只是为了增加间接编址，就需又用掉DJS-200操作码码点数（256）的四分之一，这个代价是高的。

寻址方式位法则不是由操作码指明，而是由寻址方式字段指明编址的方式，如PDP-11

的指令格式之一是：



R 为寄存器号，由“寻址方式”字段指明各种编址方式：如 (R) 是操作数，(R) 是操作数地址（一次间接）；变址型；变址间接型等等。

“操作码指明”法和“寻址方式位”法在已有机器中都广泛采用，虽然后一种方法灵活性好，但它需有专用的寻址方式字段，就它的“操作码”和“寻址方式位”的总位数来看，可能会比前一种方法的“操作码”位数要长。

指令格式的一般问题就不再讲了，下面集中于讲述指令格式的优化，即如何用最短的位数来表示指令的操作信息和地址信息。先讲述 Huffman 码制压缩的基本概念，而后着重分析操作码的优化表示。

3.1-1 Huffman 压缩概念

Huffman 压缩的概念原是对代码的压缩，但它还可应用于程序和数据结构的压缩及存储体系的设计等方面。

我们以二——十进制数的代码压缩为例来说明。

大家知道，两位二——十进制代码 (x x x x x x x x) 可以表示从 00~99 的十进制数，但是这八位代码的二进制组合本可以表示从 0~255 的数。可见，二——十进制表示具有众多的冗余码点。那么，能否压缩表示空间以减少其冗余码点呢？下面先分析这 8 位二进制位在表示 00~99 时的应用情况，见表 2.6 所示。

表 2.6 用二位二——十进制位表示 00~99

a	b	c	d	e	f	g	h
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1

看出，在第一数位 a b c d 列中，若 0~9 的出现概率一样，a 为“0”的出现概率为 80%。同理，在 e f g h 列中，e 为“0”的出现概率也为 80%。则在 a b c d e f g h 的各种组合中，a, e = 0, 0 的概率为 $80\% \times 80\% = 64\%$ ，同理可求得：

a, e = 1, 0 的概率为 $20\% \times 80\% = 16\%$ ，

a, e = 0, 1 的概率为 $80\% \times 20\% = 16\%$ ，

$a, e = 1, 1$ 的概率为 $20\% \times 20\% = 4\%$ 。

那么，能不能由上述的概率分布得出另外的优化表示法呢？

想法是这样的：对出现概率愈高的，用愈短的位数表示；而对出现概率很低的，则允许用较长的位数表示。这样，若用一位来区分上述概率为 64% 的与 36% 的情况，设概率高的用“0”表示，概率低的用“1”表示（这是任意的，当然也可相反取值），即 $a, e = 0, 0$ 的用这位为“0”表示，其它 36% 的情况，用这位为“1”表示，即 $a, e = 1, 0$ (16%) 的， $a, e = 0, 1$ (16%) 的和 $a, e = 1, 1$ (4%) 的，这位都为“1”。进一步，再用第二位区分 $a, e = 0, 1, 1$ 与 $a, e = 1, 0$ 情况。 $a, e = 1, 0$ (16%) 的，第二位取为“1”； $a, e = 1, 1, 0, 1$ (20%) 的，第二位取为“0”。当然，还得用第三位来区分 $a, e = 0, 1$ 和 $a, e = 1, 1$ 这两种情况。同上，设概率为 16% 的 $a, e = 0, 1$ 情况，第三位取为“0”；而概率为 4% 的 $a, e = 1, 1$ 的，第三位取为“1”。至此， $a, e = 00, 01, 10, 11$ 这四种情况所对应的代码应为：

$a, e = 0, 0$ (64%) 为 0
 $a, e = 1, 0$ (16%) 为 1 1
 $a, e = 0, 1$ (16%) 为 1 0 0
 $a, e = 1, 1$ (4%) 为 1 0 1

看出，为表示这四种状态似乎所需的位数反而变宽了，因为有两种情况均需要三位。然而，若从概率密度算出的平均宽度看，则 Huffman 码的平均宽度

$$\begin{aligned} &= 1 \times 64\% + 2 \times 16\% + 3 \times 16\% + 3 \times 4\% \\ &= 0.64 + 0.32 + 0.48 + 0.12 = 1.56 \text{ (位)} \end{aligned}$$

比通常表示 $a, e = 00, 01, 10, 11$ 这四种状态所需的二位要窄。这样，当概率密度不均匀时，若采用优化技术对出现概率高的事件用最短的位（时间）来表示（处理）；而对出现概率低的事件则用较长位（时间）来表示（处理），就会导致平均位数（时间）的缩短，这就是 Huffman 压缩概念的基本出发点。

至于 b, c, d 和 f, g, h 如何表示，可以有很多种方法，这对于理解 Huffman 概念已没有直接关系。一种方法是对于 $a, e = 0, 0$ 情况，其 b, c, d, f, g, h 位照抄；对于 $a, e = 1, 0$ 情况，由表 2.6 看出： b, c 两位肯定为“0”，所以只需照抄 d, f, g, h 位，当然加上 c 位也无妨；对 $a, e = 0, 1$ 情况，已用了“1 0 0”三位代码，即“ b, c ”位置已被占据，但此时“ f, g ”状态必为“0 0”，恰好可为“ b, c ”所用；对 $a, e = 1, 1$ 情况，“ b, c ”，“ f, g ”均为 0，“ b, c ”位已被“0 1”占领，“ f, g ”位可为任意值。因此，表 2.6 可压缩成：

$a, e = 0, 0$	0	b, c, d, f, g, h
1 0	1	d, f, g, h
0 1	1	d, b, c, h
1 1	1	d, \times, \times, h

7 位

这样，就把原来的 8 位代码压缩成 7 位。显然，这种 Huffman 表示法的规整性要比二——十进制法的差，但它大大减少了码点的浪费。用二——十进制法表示 00~99 的 100 种状态，浪费了 $2^8 - 100 = 156$ 个码点；而用 Huffman 法，只浪费了 $2^7 - 100 = 28$ 个码点。同理，对

三位十进制数，若用二——十进制法，则需用 12 位表示，而用 Huffman 压缩概念，可以压缩到只需用 10 位表示，使码点浪费从 3096 个减少到 24 个。

3.1-2 操作码与指令字的优化表示

研究操作码与指令字的优化表示，对于缩短指令字的长度，对于减少程序的总位数以及对于增多指令字所能表示的操作信息和地址信息，都是很有必要的。下面着重于讲述操作码的优化表示。

由前述 Huffman 压缩概念已知，为要优化表示，需知道每种状态的出现概率，对于操作码的优化表示，就是要知道每种指令的使用频度（即每种指令在程序中的出现概率）。

下面以数值例子来分析操作码的优化。设一台机器有 7 种指令，其使用频度如表 2.7 所示。看出，各种指令的使用频度差别很大，实际情况一般也是如此。显然，为表示这 7 种指令，若用定长操作码表示，则需三位。

若各种指令的出现是相互独立的（实际情况并不都是如此，在下一小节会讲到），则由信息论的基本概念，操作码的信息源熵（信息源所包含的平均信息量） H 为：

$$H = - \sum P_i \log P_i$$

由于操作码信息是用二进制位表示，则

$$H = - \sum P_i \log_2 P_i$$

按上表数值，

$$H = 0.45 \times 1.15 + 0.30 \times 1.74 + 0.15 \times 2.74 + 0.05 \times 4.32 \\ + 0.03 \times 5.06 + 0.01 \times 6.64 + 0.01 \times 6.64 = 1.95$$

就是说，对所给的使用频度值，为表示这 7 种指令，本只需（或至少需要）1.95 位（当然是平均值）就够了。

这样，三位定长操作码的信息冗余量为：

$$1 - H/\text{操作码的实际平均长度} \\ = 1 - 1.95/3 = 0.35 \text{ 或 } 35\%$$

这个数值是相当大的。那么，如何改进操作码的表示以减少这个冗余量呢？我们试用 Huffman 编码来表示。

按照上一小节所讲，Huffman 编码的操作码可设计成如下表所示：

A 方案是全按 Huffman 编码，最长位数需 6 位；B 方案是基本按 Huffman 编码，但后 4 种状态应用一般的二进制译码，使最长位数由 6 位缩短成 5 位。A 方案的平均长度为

$$\sum P_i \cdot l_i = 0.45 \times 1 + 0.30 \times 2 + 0.15 \times 3 + 0.05 \times 4 \\ + 0.03 \times 5 + 0.01 \times 6 + 0.01 \times 6 \\ = 1.97 \text{ 位}$$

表 2.7 指令使用频度

指 令	使用频度 (P_i)
I_1	0.45
I_2	0.30
I_3	0.15
I_4	0.05
I_5	0.03
I_6	0.01
I_7	0.01

表 2.8 Huffman 编码的操作码表示

指令	P_i	操作码, OP	OP 长度, l_i	操作码, OP	OP 长度, l_i
I_1	0.45	0	1 位	0	1 位
I_2	0.30	1 0	2 位	1 0	2 位
I_3	0.15	1 1 0	3 位	1 1 0	3 位
I_4	0.05	1 1 1 0	4 位	1 1 1 0 0	5 位
I_5	0.03	1 1 1 1 0	5 位	1 1 1 0 1	5 位
I_6	0.01	1 1 1 1 1 0	6 位	1 1 1 1 0	5 位
I_7	0.01	1 1 1 1 1 1	6 位	1 1 1 1 1	5 位

(a) Huffman 编码 A 方案

(b) Huffman 编码 B 方案

同理可得 B 方案的平均长度为 2.00 位, 比 A 方案的略长。

可见, 采用 Huffman 编码法, 不论 A、B 方案, 其平均长度都比定长操作码法的三位要短得多, 而很接近于可能的最短位数 1.95 位。这使信息冗余量小得多, A 方案的只有 1.015%, B 方案的为 2.5%, 相差很小, 但 B 方案的最长位数比 A 方案的短 1 位。

虽然 Huffman 编码法的平均长度最接近于 H 值, 是最优化的编码法; 然而, 它的每个操作码的位数都不同, 即不便于译码, 更不适于实际应用。所以只能采用它的变形, 如 B 方案那样; 如果前三种状态也用二进制译码表示, 则可得如表 2.9 所示的操作码编码。它是介于定长二进制译码和 Huffman 编码之间, 具有二种 (或更多种) 长度, 但仍利用 P_i 值大的, 用短 l_i 表示, P_i 值小的, 用长 l_i 表示的思想。

表 2.9 操作码的折衷表示法

指 令	P_i	操作码, OP	OP 长度, l_i
I_1	0.45	0 0	2 位
I_2	0.30	0 1	2 位
I_3	0.15	1 0	2 位
I_4	0.05	1 1 0 0	4 位
I_5	0.03	1 1 0 1	4 位
I_6	0.01	1 1 1 0	4 位
I_7	0.01	1 1 1 1	4 位

这种表示法就是所谓扩展操作码法, 如表 2.9 那样, 使用频度高的 I_1, I_2, I_3 用二位操作码的 00, 01, 10 码点表示, 而之后操作码就扩展成四位, 用低二位的四种状态表示 $I_4 \sim I_7$ 。这种表示法的平均长度 $\sum P_i l_i = 2.20$ 位, 信息冗余量约为 11.4%, 虽然比表 2.8 的 A、B 方案大, 但仍比三位操作码法的 35% 小得多, 是一种可行的优化表示法。实际机器, 如 B-1700 等, 采用了类似的这种表示法。B-1700 的面向操作系统用 SDL 语言的机器指令的操作码有 4 位, 6 位, 10 位三种长度。高 4 位的 16 个码点中, 10 个用于表示 10 个 P_i 值最大的指令, 5 个是指明操作码为 6 位长, 而最后一个指明操作码是 10 位长。这种扩

展操作码表示法和定长操作码（8 位）法以及 Huffman 编码法的比较见表 2.10。显然，B-1700 的“扩展操作码”设计得较好，它使整个操作系统所用指令的操作码总位数比一般定长法减少 39%，而又接近于 Huffman 编码法。

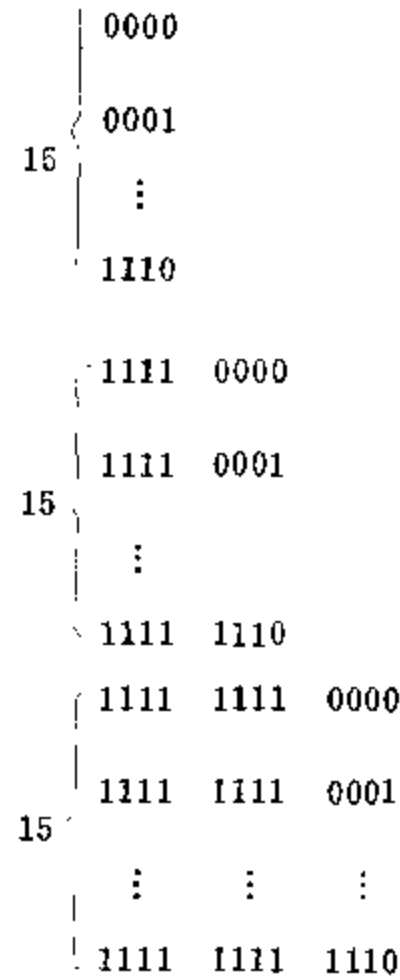
扩展操作码宜于等长扩展，如 4—8—12 位。这里可以有多种编码方法，如 15/15/15...

表 2.10 B-1700 操作码编码方式比较

编码方式	整个操作系统所用 指令的操作码总位数	改进百分比 (%)
定长 8 位	301, 248 位	0
4—6—10 位	184, 966 位	39
Huffman	172, 346 位	43

...法和 8/64/512 法等。15/15/15.....指的是四位的 16 个码点中，用 15 个表示最常用 15 种指令，用 1 个表示扩展到下一个四位；而第二个四位的 16 个码点也是如此用法，表 2.9 就是这种编码。8/64/512 指的是用头四位的 $0 \times \times \times$ 表示最常用的 8 种指令；接着，操作码扩展成二个四位，用 $1 \times \times \times 0 \times \times \times$ 的 64 个码点表示 64 种指令；而后，再扩展成三个四位，用 $1 \times \times \times 1 \times \times \times 0 \times \times \times$ 的 512 个码点表示 512 种指令。图 2.14 为这二种编码法的具体码点。

15/15/15 编码法



8/64/512 编码法

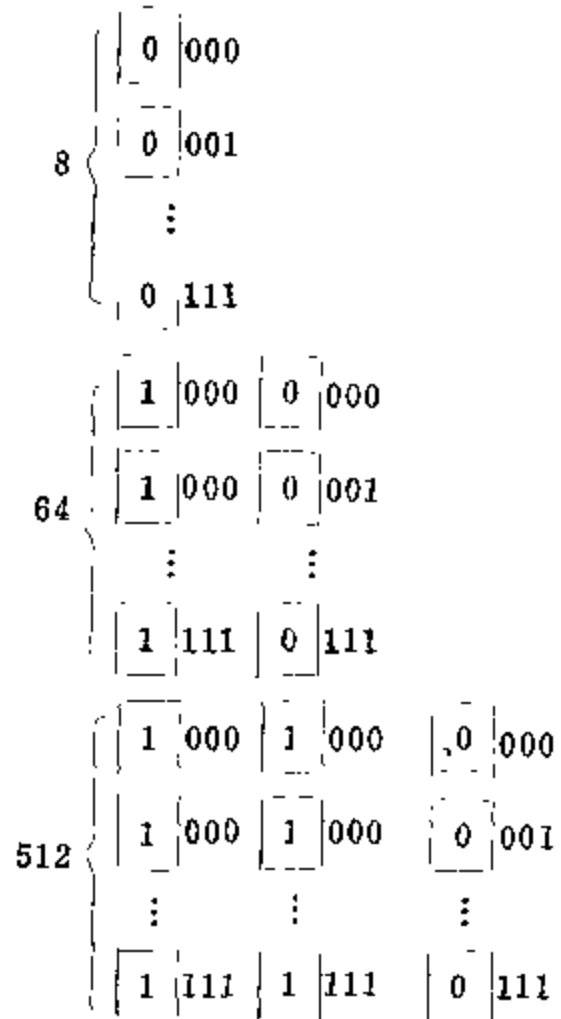


图 2.14 15/15/15 和 8/64/512 编码法

选用哪种编码取决于 P_i 值的分布。看出，15/15/15.....法可用首四位表示最常用的 15 种指令；而 8/64/512 法用首四位只能表示最常用的 8 种指令，但它用前二个四位却总共能表示常用的 72 种指令，而 15/15/15.....法用前二个四位却总共只能表示常用的 30 种指令。

所以，若 P_i 值在头 15 种指令中都比较小，但在 30 种指令后急剧减少，则宜于选 15/15/15……法，但若 P_i 值在头 8 种指令中较大，且之后的 64 种指令的 P_i 值也不是过小，则宜于选 8/64/512 法。总之，衡量的标准是看哪种编码法使平均长度 $\sum P_i l_i$ 最短。

讲这段操作码优化表示的主要目的是说明计算机中存在各种各样的信息冗余，如何尽可能地减少各种信息冗余是系统结构设计者的主要任务之一。

当然，只是有了操作码的优化表示，而没有在地址码表示和编址方式方面下功夫，采取相应的措施，程序所需总位数还是难于减少的。

如果主存是按位编址，而且指令字不受上一节所述整数边界的约束，可以一条一条紧挨着存贮，如图 2.15 所示，那操作码的优化表示是会直接带来程序所需总位数的减少。然而，在本章 § 2.1 已经讲过，这样一来，有些指令（如图中的 $K+3$ ， $K+5$ ， $K+8$ ， $K+11$ ， $K+14$ 等）却需二个主存周期才能读出，这会使机器速度明显下降（虽然主存宽度的加宽会减少这种指令字跨主存边界存贮的出现概率）。

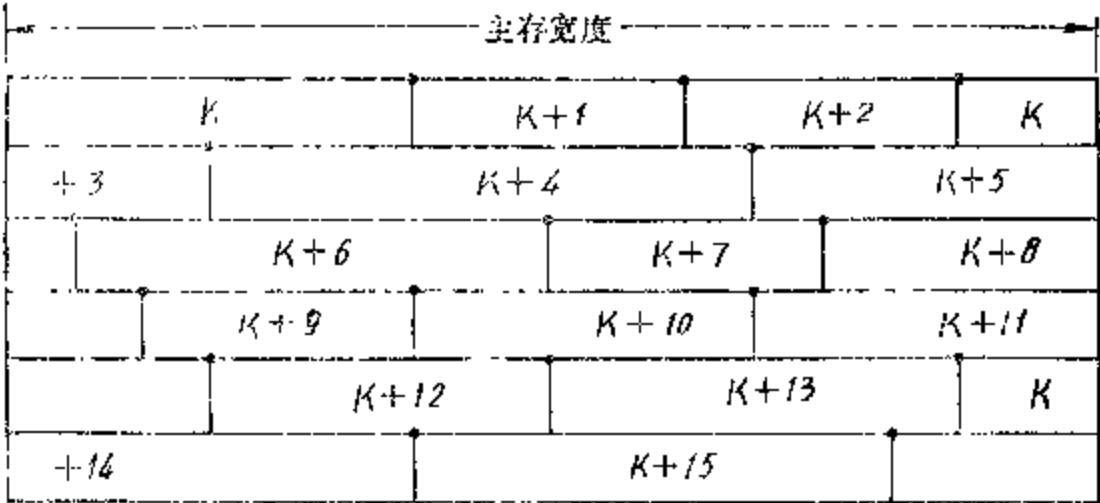


图 2.15 任意长指令字在按位编址主存中存贮的情况

那么，在维持指令字按整数边界存贮的前提下，如何发挥操作码优化表示的作用呢？我们先分析只有一种长度（ L ）的定长指令字情况。

由于操作码是优化表示，其长度 l_i 是随不同 P_i 而有多种值，即留给地址码的长度 $L-l_i$ 是可变化的。显然，若所有指令的地址码都取成一样长，那它的长度只能等于或小于 $(L-l_{i\max})$ ，这里， $l_{i\max}$ 是最小 P_i 的操作码长度；这样，对于常用的那些指令，由于操作码优化所带来的 l_i 的缩短，只是带来指令字内出现空白浪费（冗余），如图 2.16 所示。显然，只有地址码也是可变长，具有多种长度，才能利用得上空白部分。

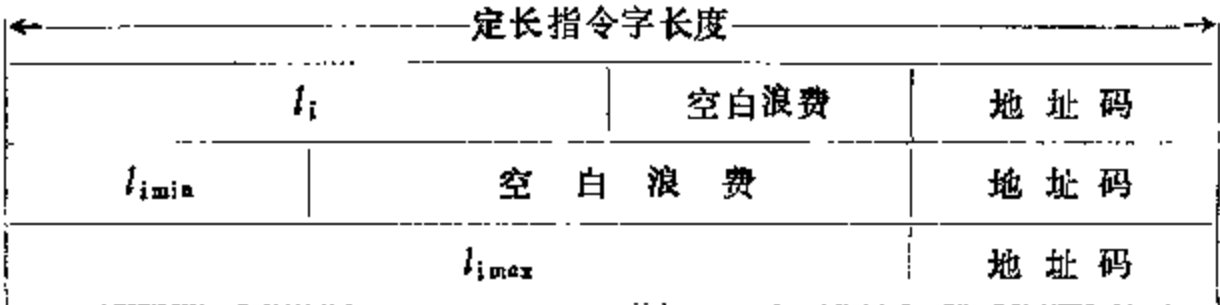
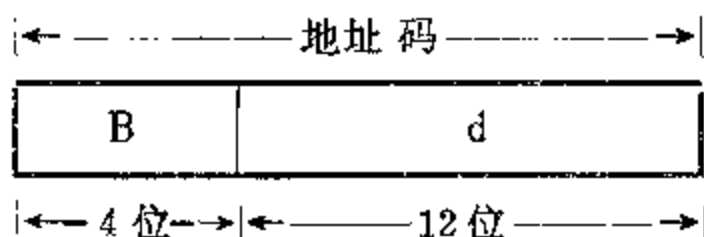


图 2.16 等长地址码发挥不了操作码优化表示的作用

我们先分析指令地址码的组成。大家知道，每个操作数访存地址的宽度，原先是根据实存空间编址的需要而定，对应于所定的最大主存容量。这样，对相同的主存总位数，当然是按位编址的地址码最长，按字节编址的次之，按字编址的最短。然而，到了六十年代以后，访存地址的宽度却已变成是按程序空间编址的需要来定，尤其是在采用虚拟存贮系统之后，

程序空间可比实存空间大得多（详见第五章），情况更是如此。

当然，不一定要求指令地址码的宽度需达到访存地址的宽度。例如，IBM370和DJS-200的访存地址都是24位宽，但指令地址码只需有16位宽：



而通过 $(B)_{8 \sim 11} + d$ 形成24位宽的访存地址。

如果有加界（基址）要求，则或是如IBM370那样，增加一个X字段指明基址寄存器号，使地址码的宽度再增大四位；或是如DJS-200那样，基址寄存器号（下界号）是由 $(B)_2$ ，或 $PSW_{3,4}$ 指明，而不必增长地址码。如果各种编址方式是由“寻址方式位”指明，则指令地址码还得加上寻址方式字段。

即使是不经变址操作形成访存地址，地址码的宽度也仍然可变。例如，对相对编址，既然访存地址是由指令地址和相对地址相加形成，而且其相对位置不会相隔过远，那需由指令地址码指明的相对地址当然不必具有访存地址的宽度。又如，对转移型指令，并不总是要求能转向程序（或主存）空间的任意位置；若把程序（或主存）空间等分为段，则访存地址可分为段号和段内地址两部分：



由于在循环内的转移多是在同一段内的转移，这时就可不必指明段号；而当要转移到别的循环时，有时只要能转移到该段的始点（对应于段内地址为零处）就可以。显然，对这二种转移，指令地址码的宽度可比整个访存地址小得多，只需等于 $MAX_1(\text{段号}, \text{段内地址})$ 。

上面讲的是为实现访存操作，指令地址码所可能具有的各种格式和宽度；如果操作数是存在寄存器内，或是经寄存器实现相对寻址（即寄存器的内容是访存地址），则指令地址码的宽度只需窄到能指明寄存器号就行。

从上述对指令地址码组成的分析中看出，一个操作数的地址码是可有有很宽的变化范围；如果安排得当，是可以和可变长操作码相配，就是对定长指令字也能显著减少其空白浪费。如果再考虑到各种指令所需的操作数个数又是不同的，则指令地址码总长度的变化范围就更宽。这样，就更能和操作码的优化表示相结合，构成冗余（浪费）量尽可能少的指令字。

例如，对图2.16的空白浪费，若该图中的地址码宽度是对应一个操作数的，则可在空白处设置别的操作数的地址，使得指令系统具有多种地址制，如图2.17所示（图上所给位

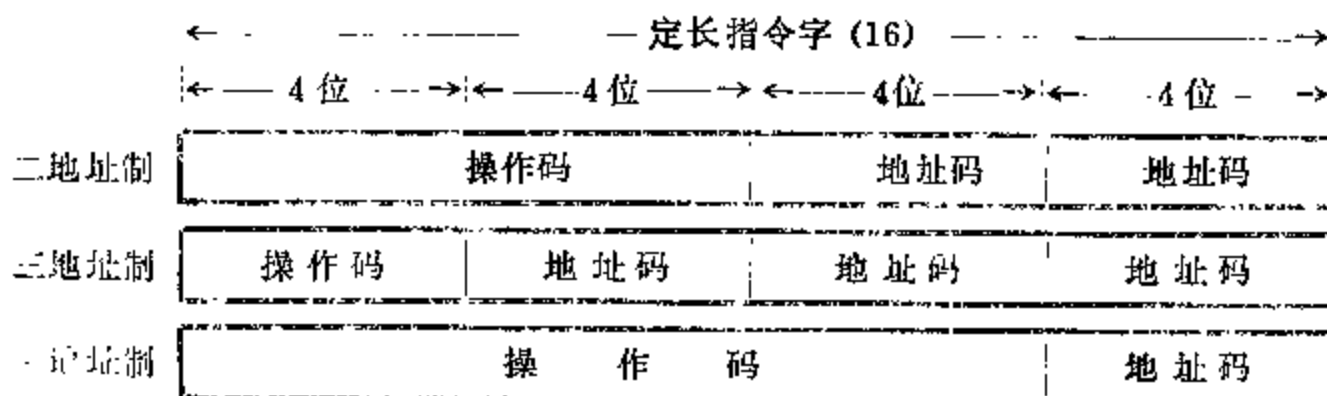


图 2.17 在定长指令字内实现多种地址制

数是随意的)。

大家知道,一条指令内的地址数愈多,程序所需的指令条数则愈少;而在这里,正是最常用的指令按前述优化表示,其操作码最短,地址数愈多,所以就更能缩短程序的长度。若操作码具有如图 2.17 所示的 4 位、8 位、12 位三种长度,且采用前述 8/64/512 编码方法,则可表示的指令数达 500 多种,肯定可以满足指令系统的需要。由于各种指令所需的操作数个数本来就不同,如果设计安排得好,那图 2.17 的多种地址制是能满足指令系统需要的。而如果指令格式是采用定长操作码法,则指令字的长度就需等于操作码的长度(只要多于 256 种指令,就需 9 位)加上最长地址码的长度($3 \times 4 = 12$ 位),达 21 位,比图上的 16 位长得多。

当然,还可以有别的方式来利用图 2.16 中的空白。例如,若该图中的地址码是指明寄存器号(用 R 表示),则空白处还可安排成如图 2.18 所示。这样,就可以有二种双地址制:RR 型和 RS 型。另外,还可利用空白处存直接操作数或常数等。

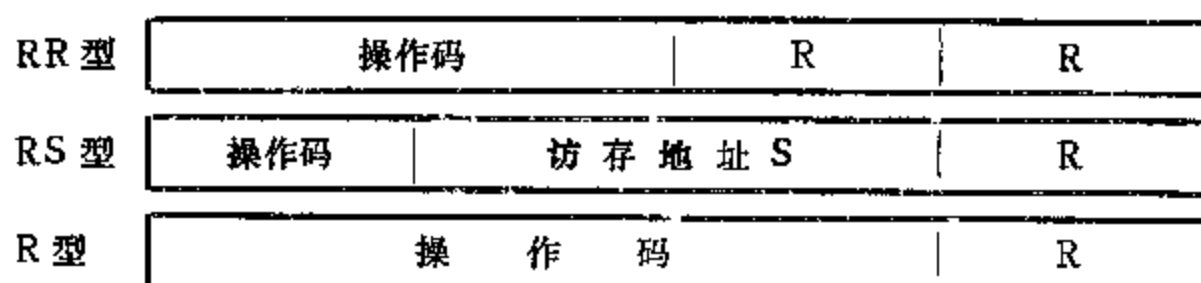


图 2.18 在定长指令字内兼有 RR 型和 RS 型

总之,操作码和指令字的优化表示,方法是多种的,效果也是明显的。虽然这些优化表示技术在小型机和微型机中广泛采用(因为小、微型机的字长短);然而不能说它们不适用于大、中型机。因为,不论对大、中、小、微型机,都需要尽可能减少机器内的各种信息冗余量。而且,到了今天,采用多输入端与门,可扩展操作码的译码时间是可以做到与定长操作码的一样;另外,若采用 PLA,则可扩展操作码法的微程序入口地址也不会难于形成。还有,可扩展操作码的码点数上限要比定长(l)操作码的 2^l 值大得多,所以可扩展操作码更能灵活地适应扩大指令系统的需要。

其实,IBM370 虽然采用的是 8 位定长操作码,但对某些指令(尤其是原 360 指令系统中没有的),如页面清除、访问方式位的复位、绝对时钟置定等特殊指令,其操作码也是由 8 位扩展到 16 位。还有,IBM370 的地址码总长需有多种长度,如图 2.19 所示。L 指明字

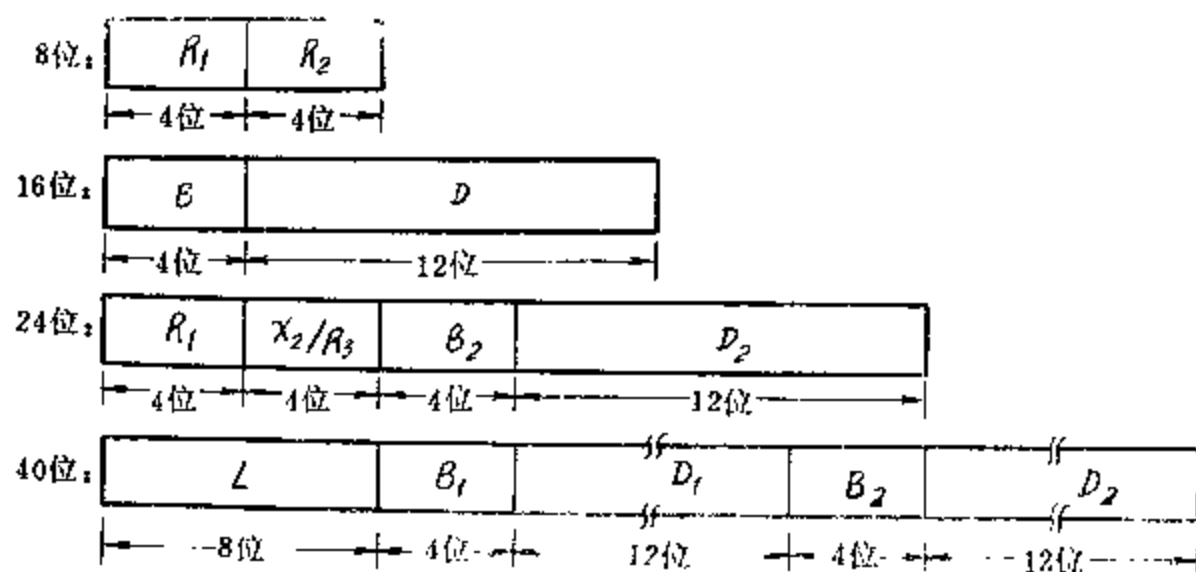


图 2.19 IBM370 的各种地址码

符行长度（字节数）。地址码格式有一地址（16 位）制、二地址制（8 位、24 位或 40 位）和三地址制（24 位）等。但由于操作码几乎都是 8 位长，所以 IBM370 的字长就有 16 位、32 位和 48 位三种。对 48 位长指令可能需二次访存才能取得出来，由于这种指令是字符行指令，在每条指令的执行过程中本需多次访存取字符，因此为取指令所增加的访存次数仍只占指令访存总次数的很小一部分。

总之，具有多种指令字长度的指令系统比只有一种长度的更能减少信息冗余量，缩短程序的长度。

3.2 指令系统的分析

本节首先简单介绍使用广泛，且有很大影响的 IBM370 指令系统；接着讲述指令系统的某些改进途径；最后讲述指令系统的可能发展。

3.2-1 IBM370 指令系统简介

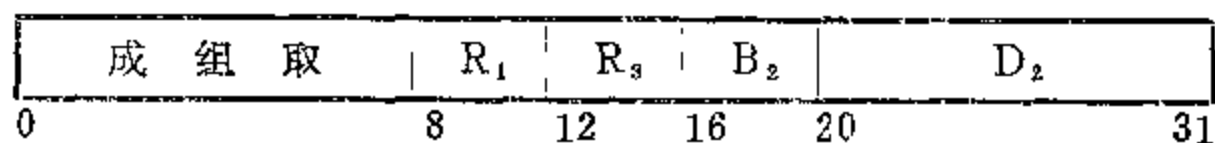
为了能利用 IBM370 的软件，IBM370 指令系统还被引用于其它多种系统，如 IBM3030 系列，IBM4300 系列，美国的 Amdahl470，日本的 M 系列和西德的西门子 7600 系列等。

IBM370 有定点数、逻辑数、浮点数、十进制数和字符行等数据表示。它的指令系统是 IBM360 指令系统的扩展，具有算术运算型、逻辑运算型、传送型和控制型等。

算术运算型指令能对定点数、浮点数和十进制数进行加、减、乘、除运算；能对逻辑数进行加、减运算；能对定点数、逻辑数和十进制数进行移位操作以及对浮点数进行除以 2 运算。算术运算型指令是两地址制，运算结果送回原存第一操作数的位置。按运算结果形成条件码，条件码有二位，可反映四种运算结果；例如，对加法指令，条件码的 00、01、10、11 四种状态分别对应于“和为零”、“和小于零”、“和大于零”和“溢出”。条件码作为程序状态字 PSW 的组成部分，可经“读 PSW”指令读出；另外，条件转移指令也可按条件码状态转移。

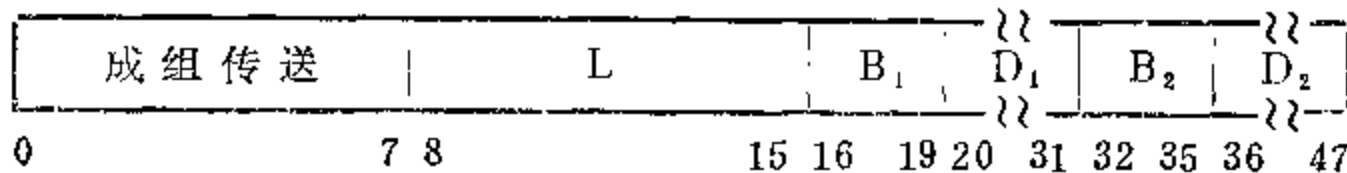
逻辑运算型指令能对字和字符行进行逻辑“与”、逻辑“或”和“按位相加”运算；能对逻辑数、定点数、浮点数、十进制数、单字节和字符行进行比较操作，并按比较结果建立条件码，这和一般机器的比较指令没过多的差别。

传送型指令除了有一般的各种单字节、单字的存、取操作和字符插入外，还有对向量传送提供某种支持的存、取指令，即能用一条指令实现字向量的存、取，其取指令的形式如下：



它把始于第二操作数地址单元的字向量读入从 R₁ 始至 R₂ 终的顺序通用寄存器中去。还有与它相对应的“成组存”指令。

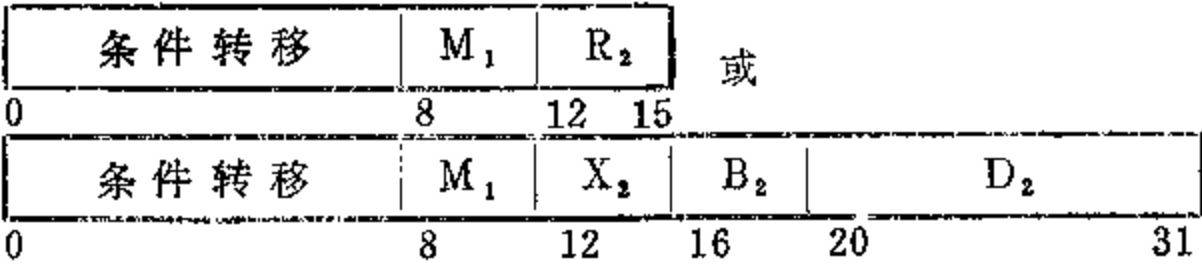
另外，还能用一条指令实现字符行（向量）传送，其指令形式如下：



它把始于第二操作数地址单元，长度由 L 指明的字符行（向量）置入由第一操作数地址单元

始的顺序单元内。这些支持向量传送的指令对于缩短程序长度有好处，也为汇编语言程序的设计提供方便。然而，只有这些指令，当然不能说 IBM370 有了向量数据表示，更何况 IBM370 没有任何字向量的算术和逻辑运算指令。

IBM370 控制型指令中的转移型指令比较灵活和完善。它的条件转移指令可以按条件码的四种状态之一转移，其指令形式为：



由 M₁ 字段决定按条件码的哪一种状态转移。M₁ 字段有四位，从左到右分别对应条件码的 00、01、10、11 四个码点；例如，若 M₁ 字段为“0100”，且条件转移指令在前述加法指令之后，则是按“和小于零”转移。转向去址为 (R₂) 或是由 X₂、B₂ 和 D₂ 形成。370 为转子程序设置的“转移与链接”指令，既执行转移到子程序的首地址，又把主程序的返回地址存于寄存器。这样，若在子程序的末尾设置上述“条件转移”指令中的第一种，就能自动返回到主程序。

IBM370 设置了“大于转移”指令，即用一条指令实现图 2.20 所示的操作。这里有 I、m_i、m_j 和转向去址等四个参量，但 370 的地址码格式至多只有三个地址，所以有一个参量 (m_j) 的地址需隐含指明。其指令形式为：

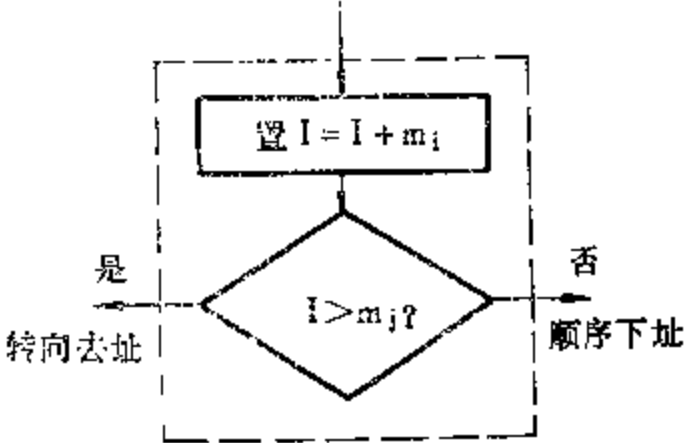
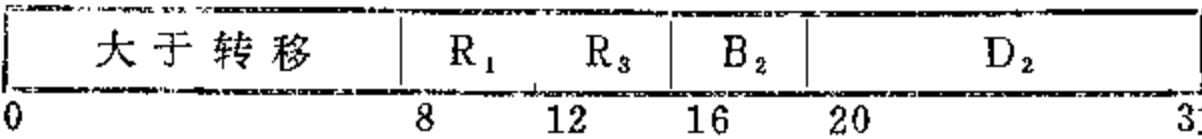
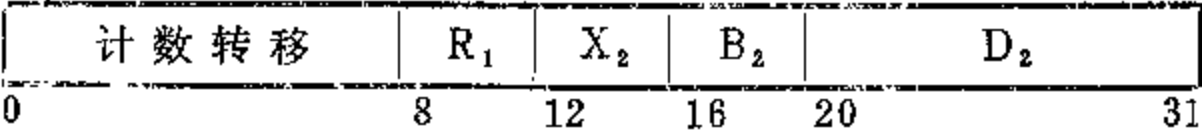


图 2.20 IBM370 的“大于转移”指令

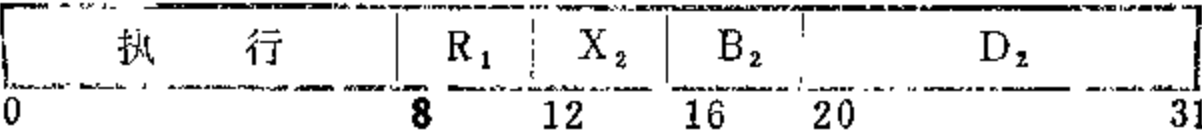


(R₁) = I、(R₃) = m_i、(R₃ + 1) = m_j、转向去址由 B₂、D₂ 形成。另外，还有“小于、等于转移”指令，除条件改为“ $I \leq m_j$?”外，其它一样。对某些循环控制，m_i = -1，条件为“ $I \neq 0$?”，此时就不必指明 m_i、m_j 值，370 的“计数转移”指令就是如此，其指令形式为：



(R₁) = I，由 X₂、B₂、D₂ 形成转向去址。这样，转向去址就可以是加基址值形成。

在上一节已讲过，在程序执行过程中，不许修改指令对于程序的检查和校验以及流水、重迭技术的应用等都有好处；但在某些情况，若能对个别指令的内容进行某种修改，是会有利于提高程序的通用性以及便于程序的编制。IBM370 为此设置的“执行”指令就既能满足指令内容可修改，但又不破坏程序内指令不许修改的约束。控制型指令中的“执行”指令，其形式如下：



当机器遇到这条指令时，按第二操作数地址（由 X_2 、 B_2 、 D_2 形成）单元内存的指令执行。

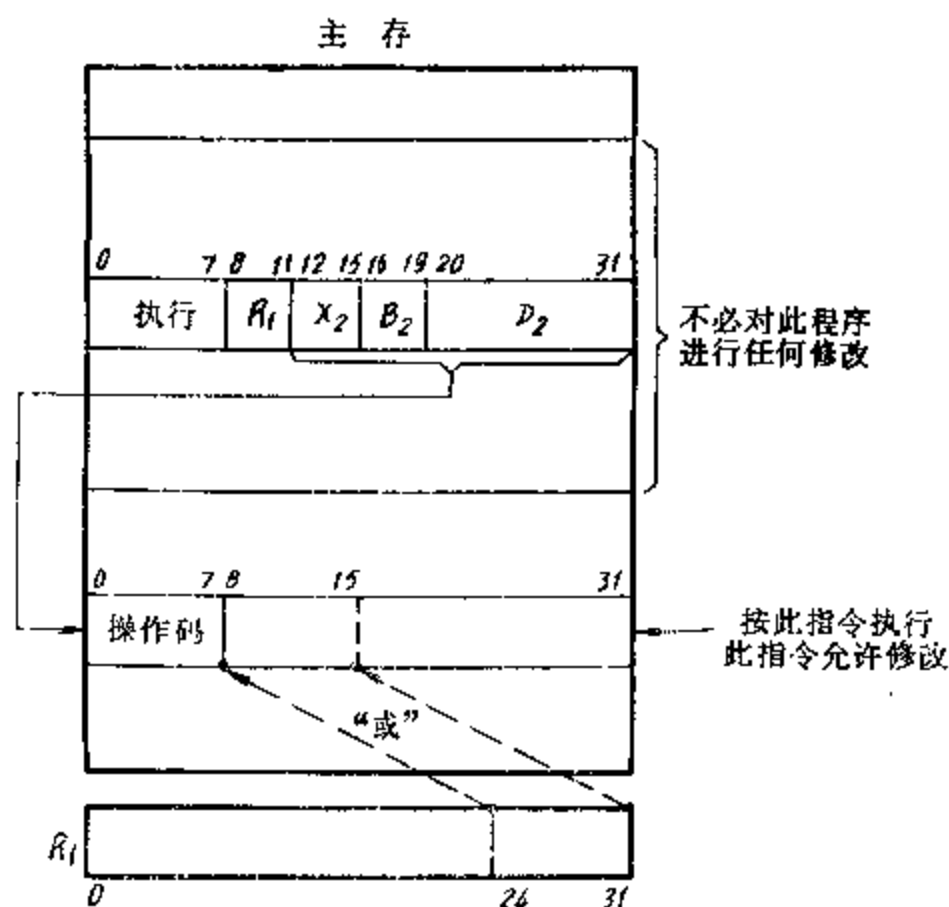
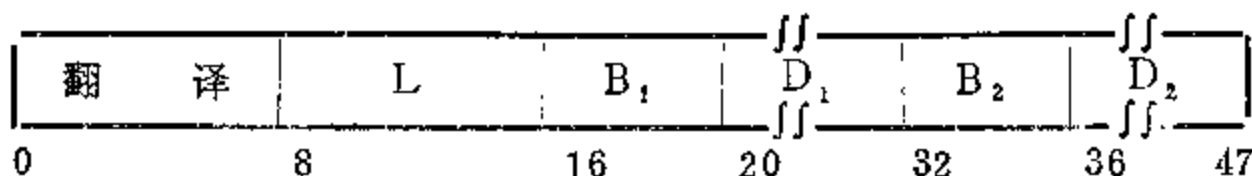


图 2.21 IBM370 “执行”指令的执行

“编辑”指令和“翻译”指令等。下面举“翻译”指令为例，它的指令形式为：



此指令可用于码制变换，如 EBCDIC 码到 ASCII 码的变换等。被变换字符存在由第一操作数地址指明的顺序单元，其个数由长度字段 L 指明。第二操作数地址是变换表的起始地址 A 。就十进制数来看，EBCDIC 码和 ASCII 码如表 2.11 所示。

看出，为把 EBCDIC 码变换成 ASCII 码，其实只需把高位的“1111”（F）变换成“011”（3）就行。一种办法是以被变换符（称为源符）作为地址，而此地址单元内存的是

表 2.11 十进制数的 EBCDIC 和 ASCII 表示

十 进 制 数	EBCDIC	ASCII
0	11110000	0110000
1	11110001	0110001
2	11110010	0110010
3	11110011	0110011
4	11110100	0110100
5	11110101	0110101
6	11110110	0110110
7	11110111	0110111
8	11111000	0111000
9	11111001	0111001

对应的变换成的符(称为目的符)，当然，变换表应在执行此指令前如下构成(用字符表示)：

地 址	单 元 内 容
A + F0	30
A + F1	31
A + F2	32
A + F3	33
A + F4	34
A + F5	35
A + F6	36
A + F7	37
A + F8	38
A + F9	39

现在，若要把 EBCDIC 码表示的 1980 变换成 ASCII 码表示，“1980”是存在第一操作数地址 Q 指明的顺序单元，则这四个单元变换前、后的内容如下：

地 址	Q	Q + 1	Q + 2	Q + 3
内 容	F1	F9	F8	F0 (变换前)
	31	39	38	30 (变换后)

此指令所执行的动作如下：先取出 (Q)，以 A + (Q) 访存取出的就是目的符，再存回 Q；而后依次再取出 (Q + 1)，(Q + 2)，(Q + 3) 进行同样的变换。

IBM370 指令系统的一个很有意义的特点是设置了用户程序不准使用而只是操作系统或其它系统软件才能使用的所谓特权指令以及相应地把机器的运行状态划分为“管(理)态”和“目(的)态”(别的机器有时称为监督态和程序态、核心状态和用户状态)。机器处于管态指的是机器正在运行操作系统，而处于目态指的是正在运行目的程序。只有特权指令能改变操作系统的状态，改变各用户对机器资源的使用状况等，这对于保证操作系统的正确、可靠运行和防止某个用户损坏系统软件以及别的用户程序等很有好处。

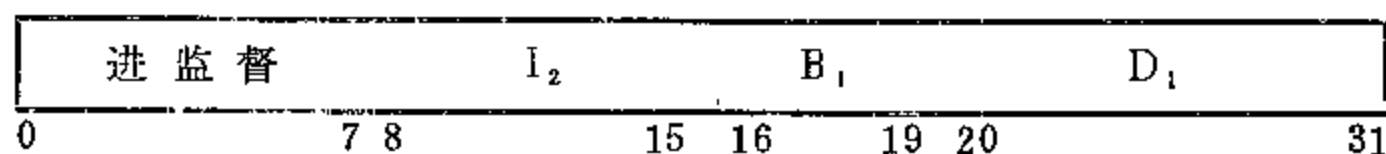
指令系统划分为只能由操作系统使用的特权指令和可由用户使用的非特权指令，在概念上可以看成是在计算机系统内配置了操作系统机器和用户机器，虽然这二类机器并不是分别由各自的硬件构成，而是重迭于同一个宿主主机上交替执行。在管态时，机器既能运行特权指令，也能运行非特权指令。即操作系统的实现是既基于面向它的特权指令，也基于并不面向它而是面向算题需要的非特权指令。

当机器在目态时，不准使用特权指令，若此时程序中出现特权指令，则机器硬件能禁止该指令的执行，并发出相应的中断。目的(用户)程序要调用操作系统需通过调管(进管、访管)指令，它当然不是特权指令。遇此指令，CPU 进入中断；交换新、旧 PSW，由此指令的地址码形成中断字的一部分，控制进入相应的管理程序。显然，机器在运行管理程序时，也可通过使用这条指令调出另一个管理程序。这样，当某个任务要调用另一个任务时，一般不是用目态指令直接调用，而是经“访管”指令进入中断，再由操作系统控制其调用。

这有助于保证多道程序的正确运行，是 CPU 共享所必需的。

特权型指令包括对存贮管理用的各种状态（如页面的存贮保护状态和其它状态）的置定、修改和读出；对页面的清除和联接；对寄存器和程序状态字的置定、修改和读出；对各种时钟（分、时、日的绝对时钟和相对时钟）值和时钟比较值的置定以及调用内部诊断程序等。此外，控制输入、输出指令（在第四章再讲）也是特权型指令。

IBM370 的“进监督”指令（属非特权指令）也有特色，其形式如下：



370 有 16 种“监督”，分别对应第 8 号控制器的 16~31 位，由 I_2 字段的低 4 位指明。若被 I_2 指明的位为“1”，则发“进监督”中断，使机器进入处理这一种监督的程序；若为“0”，则不发中断，还按原有顺序进行。通过在程序的相应点设置“进监督”指令进行跟踪，能够指明当前执行的是哪一个程序，能够统计出某段程序的使用频度，能够求得某一段程序的执行时间是多长等等。

我们在以后的章节中，结合有关内容还要讲述某些功能特殊的 IBM370 指令。

3.2-2 指令系统的改进途径

指令系统是软、硬件交界面的主要组成部分，指令系统的设计是系统结构设计的核心。可以这样说，指令系统设计中的一个主要方面是确定哪些基本操作宜于由一条指令实现、哪些基本操作宜于由一串指令实现。既然在指令系统很丰富、很完善的机器上实现的操作，从原理上讲，在只有简单指令系统的机器上照样能实现，只不过程序和执行时间会过长（当然可能会长到超过机器的实际可能），因此问题就在于实现效率如何。衡量实现效率的高低主要看各种基本操作是否易于由这种指令系统实现，实现速度是否有提高（如访存次数是否得到减少等）以及程序的长度（总位数）是否有缩短等。

循此，指令系统的改进途径中，一种是对已有机器的机器语言程序及其执行进行统计分析，寻找改进的途径；另一种是“由上往下”，围绕高级语言（包括操作系统用高级语言）和操作系统的优化实现来改进。下面先讲头一种途径。

对已有机器的机器语言程序及其执行进行的统计包括两个方面。一是统计出每种指令的使用频度；二是统计出各种指令串的使用频度。

在本章 § 3.1-2 节我们提到过指令的使用频度，本来它应有动、静之分。静态使用频度是对各种程序（如系统程序和编译成的目的程序等）所用指令进行统计得出的百分比；动态使用频度则是在程序执行过程中，对每种指令的使用状况进行统计得出的百分比。按静态使用频度来改进指令系统是面向于减少程序所需存贮空间，按动态使用频度来改进，则是面向于减少程序的执行时间。然而，大量统计表明动、静态使用频度很相近，所以我们可选用其中一种来分析，其效果会是既减少程序所需存贮空间，又减少程序的执行时间。表 2.12 是根据在 IBM360 上运行的 19 个程序，统计出最常用几种指令的使用频度。

根据各种指令的使用频度，我们可以对最常用的指令，提高它们的功能，加快它们的速度，缩短它们的指令字长等等；而对使用频度很低的，则可考虑取消之。

至于统计出各种指令串的使用频度，则是为了在指令系统中设置新的指令，对最常用的指令串，用新的一条指令去替代其中的一串。

表2.12 IBM360指令的动、静态使用频度

指 令 类 型	静 态 使 用 频 度 %	动 态 使 用 频 度 %
L (取)	28.6	27.9
ST (存)	15.0	9.8
BC (条件转移)	10.0	13.7
LA (取地址)	7.0	6.1
SR (减)	5.8	4.5
A (加)		3.7
C (比较)		6.2
SLL (逻辑左移)	3.6	
BAL (转移与链接)	5.3	
IC (插入字符)	3.2	4.1

下面举一些例子来说明。

例如，下面这一对指令：

SR (减) 寄存器、寄存器

IC (插入字符) 寄存器、主存

在 IBM360/370 程序中经常出现。它用于经 SR 清除某个寄存器，而后经 IC 由主存取来一个字符送进这个寄存器。若我们用一条“清除寄存器并装入字符”新指令来替代这一对指令，则这条新指令的完成时间几乎和 IC 指令的没有差别。有人预期，这个改进会使程序的平均长度缩短 1%，程序的执行时间会减少 1.5%。

又如，IBM370 的编译程序经常形成下述指令串：

ST (存)	REG, SAVE	(REG)→SAVE
L (取)	REG, VARIABLE	(VARIABLE)→REG
LA (取地址)	REG, N(, REG)	(REG)+N→REG
ST (存)	REG, VARIABLE	(REG)→VARIABLE
L (取)	REG, SAVE	(SAVE)→REG

它用于给 VARIABLE 单元内的数增 N。由于 IBM370 的所有字相加运算只能在寄存器之间，或寄存器与主存之间进行，且运算结果只能存在寄存器，而通用寄存器又只有 16 个，因此需把 REG 寄存器原存内容先暂存于主存，而后才能用它实现 (VARIABLE) 加 N。LA 指令是把由 X, B, D 形成的地址存入寄存器，若此寄存器与变址寄存器是同一个寄存器，且 D=N，则通过变址相加实现加 N 操作。这五条指令共占 20 个字节，在 370/145 上要化 8.8 微秒才能执行完。看出，这个指令串是可用一条“增量”新指令替代。要求这条指令有直接操作数 N 和形成访存操作数地址的地址码字段。它能执行给访存操作数增 N，并把相加结果送回主存。对 IBM370 的指令格式，这条新指令只需 4 个字节，在 370/145 上，看来只需 2.7 微秒就能实现。

用新的指令来替代常用指令串的这种思路，其实包括前面讲过的设置新的数据表示、浮点指令、字符行指令、十进制运算指令等都是成功的例子。这种思路当然可以引伸于把常用宏指令和子程序（如双倍长运算、三角函数、开方、二——十进制转换和编辑、翻译等子程序）改为用一条机器指令实现。

指令系统的这种改进易于被用户所接受。因为新、旧指令系统是相似的，如果原有指令系统不删减，而只是增加强功能新指令以替代常用指令串，那软件兼容不受破坏，原有软件照样能运行，而按新的指令系统编制的程序又具有更高的效率。这种维持系列机兼容概念的改进往往是计算机生产厂所希望的。

但是，这种改进的实际效果如何，需要进行深入细致的分析。首先，常用指令串并不一定是计算机系统的“瓶颈”。例如，若“瓶颈”是在 I/O、通道系统，则不论如何替换 CPU 的常用指令串，其实际效果是不大的。因此，宜于替换的应是对“瓶颈”有直接影响的常用指令串。

另外，对于长度比较长或功能比较复杂的指令串的替代，其效果的评价就不象替代上述那二个简单指令串那样容易。对于采用微程序控制的大多数机器，这种评价主要是微程序实现和机器语言子程序实现的比较。例如，正弦函数可用

$$\begin{aligned}\sin x &= c_1 x + c_3 x^3 + c_5 x^5 + c_7 x^7 \\ &= x(c_1 + x^2(c_3 + x^2(c_5 + x^2(c_7))))\end{aligned}$$

求得。这当然可用具有加法和乘法指令的机器语言子程序实现， c_1 、 c_3 、 c_5 、 c_7 常数存在主存内；但若用一条复合（复杂）指令替代它，并经微程序实现，则可以大大减少访主存取机器指令的次数，如果 c_1 、 c_3 、 c_5 、 c_7 常数又是存在微程序存储器（控存）内，那它们的取出时间可以短得多。这些，可使微程序实现的速度比机器语言子程序的至少快 1~2 倍。然而，这段微程序却要占用比主存宝贵得多的控存空间。显然，若某个题目要频繁调用此机器语言子程序（指令串），则为它的替换所付的控存代价是合算的；而若某个题目只调用少数几次，那怕用微程序实现快得多，那也不如用机器语言子程序的合理。科研计算题目和事务处理题目所需的（所常用的）基本运算和基本操作是不同的。因此，以新指令替代指令串的这种指令系统设计，往往和机器的应用范围、所算题目关系很大，对于长指令串的替换，更是如此。在第一章的“微程序”那节已经讲过，从原理上，只需更换控存的内容，就能使机器具有面向各种应用的不同指令系统。这样，似乎尽可设置众多的强功能复合指令，将它们的微程序存在辅存，按照所解题目的需要再调入控存。然而，目前日益庞大的软件最终是要落实于某种指令系统；因此指令系统的灵活可变对于软件的设计和实现不见得有什么好处，甚至有可能造成混乱。这就是说，计算机系统是否能用得上，或是否需要二级（机器语言级和微程序级）可编所能带来的灵活性。看来，要用得上这种灵活性，还得在理论上和实践上作很大的努力。

不过，近年来把操作系统中用得频繁且对速度影响大的子程序进行硬化和固化，已取得了明显的进展，这就是把操作系统内用得频繁而对速度影响大的一段汇编语言程序（有人称之为过热点，HOT SPOTS）改为直接由微程序或硬件实现。它可以是以一条新的机器指令方式出现，也可以是由操作系统专用的某种扩展部件实现。前者是其它系统软件甚至包括用户程序都能使用的，后者则只是在操作系统的实现中可以用得上。虽然前者的利用率会比后者高，然而后者能以选购扩展部件出现，几乎不必对原有系统作什么修改。当然，对用户来

讲，这二种方式都是透明的。

有人称这种机器指令为管理用指令，它的功能可大可小。功能小的只是执行操作系统常用的基本操作，如对一个字内某些指定位的测试与置定、N向（多分支）转移等。相信随着硬件价格的下降，这种指令的功能会逐步扩大，例如有人提出可以设置“I/O中断分析”指令，使得通过这一条指令就能够按卡片读入机、行式打印机、磁带机和磁盘机的下述状态：通道结束、控制器结束、设备结束、控制器出现例外情况、设备出现例外情况、校验有错等等，分别直接进入相应的中断处理程序。这有利于提高常用外设的中断响应速度和减少辅助操作的时间。由于这种复杂功能的指令用固件实现比用硬件实现要合理，因此，分析设置这类指令是否合适，实质上是分析某种功能是用软件合适呢？还是用微程序实现合适。一般地说，如果这种指令的执行中需很多次访问主存那比起软件实现来说微程序实现不见得有什么好处，因为主要受限于访主存时间，使得实现时间缩短不了多少，但控存资源却比主存资源宝贵得多。

IBM近年来已用微程序实现 OS/VSI 和 VM/370 操作系统中的下述过热点：

进管处理——对要求进入前述管态、调用操作系统的申请进行分析、并按它执行N向转移，进入相应的管理程序；

I/O中断处理——对中断申请进行分析并转移到相应的中断处理程序（类似刚讲过的I/O中断分析）；

通道控制字翻译程序——对通道控制字进行翻译并传送到实存；

I/O工作区的管理——动态地对I/O工作区进行分配；

虚拟存贮管理——对不同容量要求的各个用户进行主存分配；

页面管理——在辅存与主存之间进行页面交换；

等等。

增加了这种微程序扩展部件后，操作系统的速度可提高20%以上。IBM370的138，148，158，168都逐步配上了这种部件。这种方法的好处在于它的加入对系统结构的别的方面没有影响，也不会影响到没有选购这种部件机器的操作系统的正常运行，仅仅是使有这种扩展部件的机器其操作系统的运行速度得以提高。这种用法和前述高级语言的实现中增加浮点部件、十进制部件的用法相似。IBM-3033则进一步扩大了这种扩展部件（称为系统扩展选购件）的功能。此外，IBM还提供了支持VM/370的VMA固件以加速虚拟机系统中虚指令到实指令的映象。

显然，随着器件价格的日益便宜，可读写控存的容量可大大扩大，指令字的长度也可增长；这样，上述这种对已有机器的机器语言程序及其执行进行统计分析，从指令及指令串的使用频度寻找增强、改进指令系统的途径，应是改进指令系统的一个重要方面。

下面简述第二种思路，即如何围绕高级语言和操作系统的优化实现来改进指令系统。

同样，也从各种高级语言语句的使用频度开始分析。表2.13是从高级语言源程序的分析统计中得出的各种语句的静态使用频度。FORTRAN的“其它”包括FORMAT语句的4%和CONTINUE语句的4%。“DO”这栏对COBOL是PERFORM语句，它和FORTRAN DO语句的含义相似。对COBOL的原统计中没有GO TO的数据，这当然不能说COBOL源程序不必用到它；一般地，它的比例和IF的相近，若取成一样，并相应地减少其它语句的比例，则为最后一行的COBOL（估值）。

表 2.15 DO 循环的语句数目和嵌套深度

循环内的语句数目	1	2	3	4	5	>5
所占百分比%	39	18.5	9.5	7	13	13
循环的嵌套深度	1	2	3	4	5	>5
所占百分比%	53.5	23	15	5.5	1.5	1.5

其语义如图 2.22 所示:

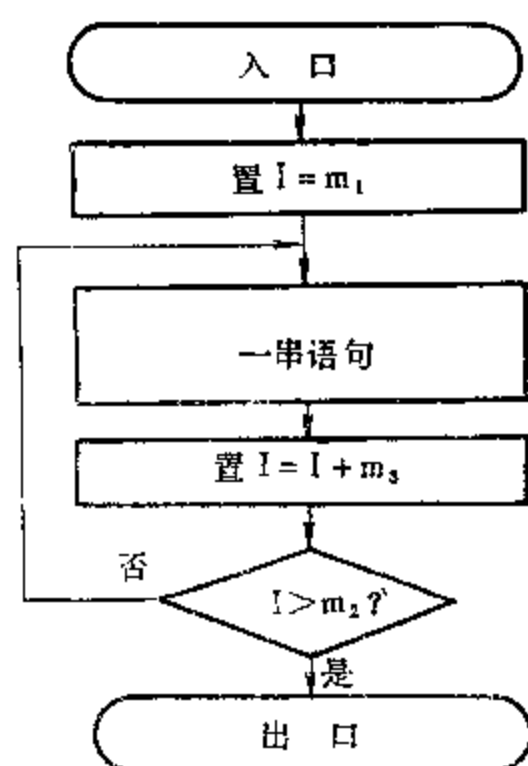


图 2.22 FORTRAN DO 的语义

图中的“一串语句”指的是在 DO 语句之后直至 n 标号的一串语句。看出, 为控制循环次数所需的辅助操作指的是对应于“一串语句”之外的那些操作。其中, 增量 ($I = I + m_3$) 和判循环次数 ($I > m_2?$) 是每个循环内都要执行。显然, 若用一条复合机器指令来实现这二个操作, 那当然比分别用“相加”指令和“条件转移”指令来实现, 能明显减少辅助操作时间。这条复合机器指令其实就是前述 IBM 370 的“大于转移”指令。包括“大于转移”、“小于、等于转移”和“计数转移”等指令的设置都是为了 DO 循环的优化实现。这种为优化实现高级语言, 从静态使用频度着手设置相应机器指令的思路日益为人们所重视。例如, 1978 年问世且近年来已打开销路的 VAX-11/780, 它是在 PDP-11 基础上发展起来并与它向前兼容的。它的指令系统中, 比原 PDP-11 指令系统增加的部分中, 就有和“大于转移”指令相同的“相加比较与转移”指令。

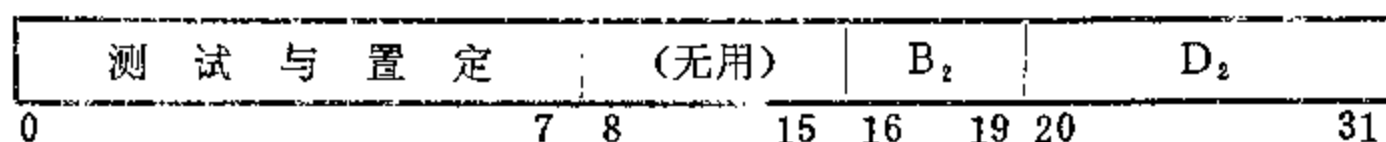
当然, 这类指令的设置要慎重, 这点我们下面要讲到。

上面表 2.13, 2.14, 2.15, 对 FORTRAN 的种种分析都是指其静态值; 对于动态值, 看来赋值语句的动态使用频度会增大, 这是因为在循环内主要是赋值语句, 而它们在程序的执行中是会多次重复的。不过, 赋值语句中的一元赋值语句的比例将会减少, 因为它们多用于循环之外, 作置定起始值用, 而循环内的多是运算型的。然而, 动、静态值还是相近的。因此, 围绕高级语言程序的各种静态特征来设计和改进指令系统还是可行的。

当然, 如何围绕高级语言的优化实现来改进指令系统, 从静态使用频度着手只是一个方面, 在下一小节以及 § 4 我们还要分析。

至于如何根据操作系统的需要设置相应的机器指令, 除了前面讲过的, 对影响速度大的常用子程序, 设置对应的机器指令之外, 往往还需设置支持操作系统的特殊指令。下面举例来说明。

IBM 360 为支持多道程序对公用区的正确使用设置了“测试与置定”(TEST AND SET) 指令。这条指令的动作内容并不复杂, 然而, 它是从系统结构对操作系统提供重要支持的一条指令。其指令格式如下:



它根据由 B₂、D₂ 所指明的单元（以字节编址）的最左一位的状态去置定条件码，然后紧接着通过把此字节单元置全“1”而把这最左一位置成“1”。这条指令在系统结构上的最主要特征是：不论机器工作于何种控制方式（如第三章要讲的重迭方式和流水方式等），必须在它之前的所有指令访问完该字节单元后（若有这种访问），“测试与置定”指令才能去访问它；而且，从访问该字节单元，到按其内容置定好条件码，直至完成置全“1”，该单元不能被其它指令和其它处理机所访问。“测试与置定”指令也不能被中断，CPU 不能在该指令未完成之前转去执行别的指令。我们可以把此字节单元的最左一位作为标志位，它的测试和置“1”是不可分隔地完成的。

下面结合对多道程序公用区的管理来讲这条指令的作用。设 K 公用区为 A、B 二个进程所公用，要求当某个进程开始使用 K 公用区，且没使用前，另一个进程不得使用它，

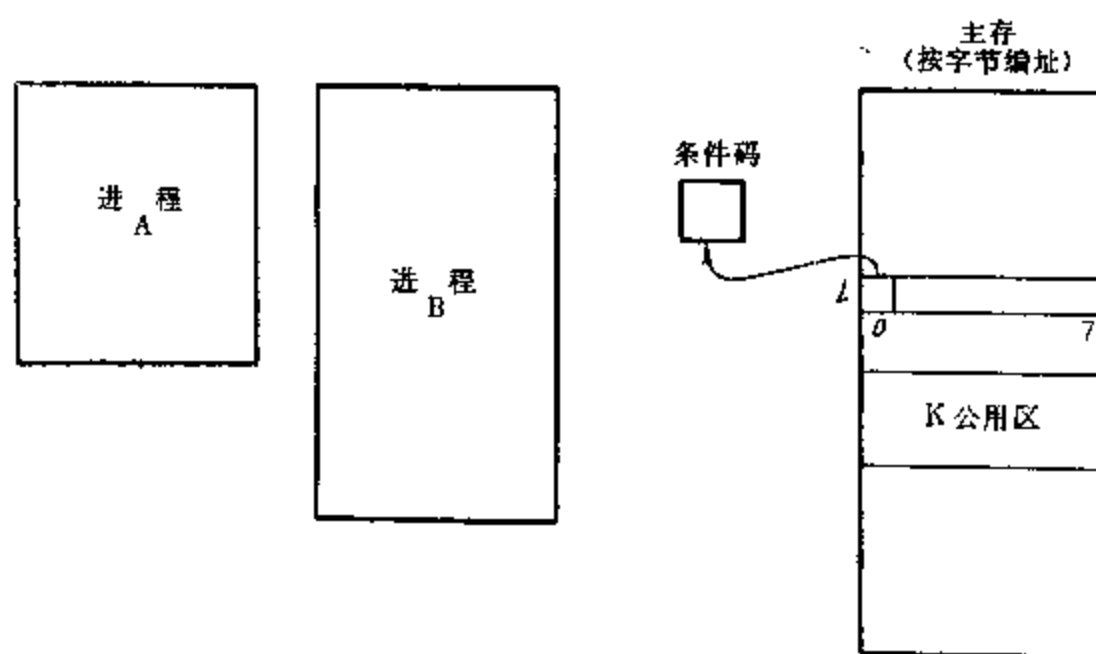


图 2.23 “测试与置定”指令的用途

以免引起混乱。用 L 单元的最左一位(L)₀ 作为 K 公用区的使用标志位。(L)₀ 为“0”，表示 K 公用区为闲置，任何进程都可使用它；(L)₀ 为“1”，表示 K 公用区正被某个进程使用，从而别的进程不能使用它。

这样，每个进程要使用 K 公用区时，必须先判定 (L)₀ 位，“测试与置定”这条指令就是用于这种判定。它用 (L)₀ 去置定条件码，从而可按条件码是“0”或“1”，决定是否可使用 K 公用区；并紧接着由这同一条指令将 (L)₀ 置“1”。若 (L)₀ 原为“0”，此动作将它置成“1”，既对别的进程封锁此公用区，直至本进程使用完此公用区，由相应指令将 (L)₀ 置“0”；若 (L)₀ 原为“1”，则由它形成的条件码控制本进程不得使用此公用区。

因此，若进程 A 用此指令判定可使用此公用区后，进程 B 就不能使用它，反之亦然，从而保证了公用区的正确使用。如果 IBM360 没有设置这条指令，那就至少需要由以下指令串：

取 L
判 (L)₀
送全“1”到 L 单元（置“1” (L)₀）

来实现对 $(L)_0$ 标志位的操作；然而，这是有可能会出错的。例如，若进程 A 正好执行完“判 $(L)_0$ ”，判出 $(L)_0$ 为“0”后，CPU 却因分时等其它原因转去执行 B 进程，则因 $(L)_0$ 仍为“0”，B 进程当然可使用 K 公用区；而若在 B 进程还没使用完 K 公用区，即 $(L)_0$ 仍在“1”状态，可是 CPU 又转去继续执行 A 进程，则 A 进程就会因为在之前已判定 $(L)_0$ 为“0”，因而也进入使用 K 公用区，从而造成混乱，产生错误。

这条“测试与置定”指令是操作系统实现 Dijkstra 信号灯的基础；不用这条指令，要保证公用区的正确使用，至少在执行上述对 $(L)_0$ 标志位操作的指令串时，机器不得响应中断。然而，这个指令串是属用户程序的进程 A 或 B，这样，就得要求能由用户程序在目态时控制对中断的屏蔽，这会引出多道程序操作系统的混乱，是不允许的。如果改为是由操作系统执行对 $(L)_0$ 标志位的操作，那是可做到在执行上述指令串时，屏蔽所有中断；然而，这就要求用户程序每当要访问公用区时，都得调用操作系统，而操作系统的进、出都需有辅助操作，都要耗费额外的时间，这又会影响解题时间。

在 IBM360 最初的指令系统中并没有这条指令；可是，很快地随着操作系统的配置就引入了“测试与置定”这条指令。这形象地说明了按操作系统需要设置相应机器指令的必要性。当然，为要实现这条指令所要求的操作，不仅要硬件保证在指令执行完之前不得被中断，还要能由中央控制器往主存控制器发出相应的控制信息，使得在对该单元的读、写访问之间，别的指令和其它处理机不能访问它。

在有了这条指令之后，可以保证公用区的正确使用；然而，在具体实践中却又发现了由它引起的，操作系统称之为“死锁”的问题。我们还是以例子来说明。

设上述 K 公用区内的 C 单元存了各个进程公用的某个计数值 X，要求各个进程都能独立地对它增值（加“1”）。

基于前面的分析，简单地在进程 A、B 设置如下的指令串：

进程 A		进程 B	
⋮		⋮	
取	C	取	C
增	值	增	值
存	C	存	C
⋮		⋮	

是可能会出现增值错误。因为，若进程 A 执行了“取 C”，但在执行完“存 C”之前，CPU 因分时等原因切换到进程 B，则经过进程 B 的增值操作，虽然 X 已增值至 $(X+1)$ ；然而，在转回进程 A 之后，它仍然按切换到进程 B 前已执行了的“取 C”取来的 X 值进行增值，从而使得经过进程 A、B 增值操作后，C 单元所存内容并不是应有的 $(X+2)$ 值而是错误的 $(X+1)$ 值。

通过引入“测试与置定”指令，构成如下的指令串（“条转”指令按条件码为“1”转移）：

进程 A		进程 B	
⋮		⋮	
LOOP A	测试与置定 L	LOOP B	测试与置定 L

条 转	LOOP A	条 转	LOOP B
取	C	取	C
增 值		增 值	
存	C	存	C
清 除	L	清 除	L
⋮		⋮	

上述增值错误是可以排除的。这里，由标志位(L)₀和“测试与置定”指令保证了只在某个进程全部完成了增值操作，清除了(L)₀后，另一个进程才能访问到C单元。然而，若(L)₀原先为“0”，进程A通过执行“测试与置定”指令取得对C单元的“访问权”，但在执行完“清除L”指令前，CPU切换到执行进程B，则进程B由于(L)₀已为“1”，从而在“测试与置定”和“条转”指令间循环。

这样，进程A占有了标志位资源，但需要取得CPU资源才能继续执行下去，而进程B正相反，占有了CPU资源，但需要取得标志位资源才能继续执行下去。就是说，机器进入了“死锁”状态，只有靠“外力”才能走出这种状态。这可能是各种外中断，也可能是进程B的时间片已用完。然而，这里可能出危险：如果进程B正好是操作系统的某个特权程序，规定必须在它执行完之后进程切换才能进行，那机器就可能进入在“测试与置定”和“条转”指令之间一直循环不止的“死胡同”。

从系统结构，指令系统可以有什么方法去解决这个问题呢？出现死锁是由于采用上述标志位技术所引起的，因此，出路在于如何既不采用这种标志位技术，但仍要保证不出现前述的那种增值错误。从根本上看，增值错误是由于执行“存C”指令时，要存进去的计数值是按错误的起始值算出的，即起始计数值实际上已由B进程修改过了，但A进程却觉察不了。因此，如果在执行“存C”时，能判定起始计数值是否仍是原先执行“取C”取来的值；如果是，则为正确增值，可以执行写C操作，如果不是，则按修改了的起始值重新增值后，再执行写C操作，那增值错误就可排除。循此思路，IBM370就设计了“比较与交换”(COMPARE AND SWAP)指令，其指令形式如下：

比 较 与 交 换	R ₁	R ₃	B ₂	D ₂
0	7 8	11 12	15 16	19 20
				31

设由B₂，D₂形成的物理地址为S₂，此指令可简写成“比较与交换R₁，R₃，S₂”，其功能如下：

(R₁)与(S₂)比较

若(R₁)=(S₂)，则(R₃)→S₂，条件码置成“00”

若(R₁)≠(S₂)，则(S₂)→R₁，条件码置成“01”，从开始访问第二操作数，直至若比较相等，完成(R₃)→S₂之前，由硬件保证S₂单元不能被其它CPU所访问，且指令不能被中断。还有，同“测试与置定”指令一样，必须在之前的所有指令都访问完S₂单元后，这条指令才能访问它。

这样，若使S₂等于上述C，且经“取C”取来的起始计数值是存在R₁，则条件码为“00”，表明起始计数值没有被别的进程修改过，从而增值结果是正确的，可以将它写入公用单元；若条件码为“01”，表明起始计数值已被别的进程修改过，从而增值结果是错误的，

需把已被修改了的起始计数值取到 R_i ，按它进行增值后，才能送回公用单元。按此，前述指令串应改写成：

	...		
	取	R_i, C	$(C) \rightarrow R_i$
LOOP A	取	R_j, R_i	$(R_i) \rightarrow R_j$
	增值	R_j	$(R_j) + 1 \rightarrow R_j$
	比较与交换	R_i, R_j, C	若 $(R_i) = (C), (R_j) \rightarrow C$ 若 $(R_i) \neq (C), (C) \rightarrow R_i$
	条转	LOOP A	若条件码为“01”，进行转移
	...		

大家知道，在某个进程（设为上述进程 A）还未进行完，切换到另一个进程（设为进程 B）时，对应进程 A 的通用寄存器内容需保存在主存，并在由进程 B 转回进程 A 时，又需将保存了的内容重新置回通用寄存器。这样，若进程 A 执行完“取 R_i, C ”，CPU 就转去执行进程 B，则在转回进程 A，执行“比较与交换”指令时，必然出现 $(R_i) \neq (C)$ 。然而，经“条转”指令转移回到 LOOP A，重新增值后，就可得到正确的增值结果。

看出，这里没有采用“标志位”技术，从而不会进入“死锁”，每个进程经过自身的循环总能继续前进。

“比较与交换”指令还可以有其它很多用法，它可以替代“测试与置定”指令，只需使 $(R_1) = \text{全 } 0, (R_3) = \text{全 } 1, S_2 \text{ 存标志位}$ 。然而，前者比后者要灵活得多。

“测试与置定”和“比较与交换”指令不仅是在多道程序的公用数据管理中很有用，在多处理机系统的信息交换与管理中也很有用，这在第七章还会谈到。

从上述二条指令的例子可以看出，在改进指令系统时，增设这种支持操作系统的特殊指令是有意义的，而且往往是必不可少的。其实，操作系统深深依赖于系统结构所提供的支持，这种情况还表现在很多方面。例如，如果没有系统结构提供的中断硬件，那么，分时操作系统和多道操作系统的实现是难以想象的。又如，操作系统存贮管理的主要功能如果不是系统结构为其提供了专用硬件，先是上面讲过的上、下界寄存器，后是第五章要讲的那些硬件，那就会无法实现或是即使能实现，其效率（速度）也会低到毫无使用价值的地步。因此，如果说高级语言可以在不考虑其实现要求的系统结构上实现，即所用的编译程序能在不考虑编译要求的系统结构上编制出来的话，那么，操作系统往往就做不到这点。因此，愈来愈多的人认为应该把操作系统和系统结构这二者结合起来进行研究，还有人提出应该把二门课合并，统一讲授，称之为“操作系统和系统结构”。

操作系统是要花费大量机器时间的。一般机器在正常算题时，可能有近一半左右的中央处理机时间用于操作系统的运行上。这样大的耗费（包括机器资源的耗费）并不是所有的人充分意识到的，尤其是计算机厂家对于操作系统带来使用上的方便性是以消耗如此大量的机器时间为代价这一点是较少提及的。在改进系统结构时一定要考虑到如何为操作系统提供更好的支持，以提高其实现效率。在选购机器时，不能只注意机器加、减、乘、除的运算速度，还要注意哪种系统结构更有利于操作系统的实现。虽然目前一般机器的系统结构是考虑了对操作系统的支持，然而这种支持是不够的。操作系统所占用的中央处理机时间比例从六

十年代初的 10% 到 15% 增加到目前的近 50% 以上,这固然跟操作系统功能的增加和复杂化、可靠性和可恢复性(如重复执行)能力的提高以及 I/O 管理的加强等等有关,然而这也跟系统结构,尤其是指令系统对操作系统的支持还不够充分有关。

我们在前面讲过,为加快操作系统的执行,途径之一是把操作系统中常用的子程序硬化或固化实现。然而,是否可以把整个操作系统的全部功能硬化或固化实现呢?七十年代初期确实作过这方面的尝试,如 Symbol 系统的全硬化,Venus 系统的全固化等,但并没有取得预期的成功,未能使计算机系统的性能有明显提高,反而降低或失去了应有的灵活性,给机器带来使用上的不便。就是说,全硬或全固不应是我们追求的目标,而软、硬结合,各发挥其所长,并合理地逐步扩大硬的比例才是可取的。在操作系统功能的软、硬实现中,经硬件直接实现是为了提高系统的执行效率,缩短操作系统所占用的中央处理机时间;而经软件实现是操作系统功能目前的基本实现方法,它对于提供系统应有的灵活性及例外、特殊情况的处理是很适宜的。因为硬件难于实现灵活性,而且要硬件对例外、特殊情况也能处理,则必然要使它过分庞大、复杂。但那些用得频繁的和基本的操作,若其功能是稳定不变而且用软件实现费事、费时的则宜于硬化实现。看来,把进程调度,存贮管理和数据访问等的基本操作硬化,设置相应的指令是可取的。

总之,不论从前述如何为操作系统提供专用指令及专用硬件的需要,或是从如何缩短操作系统所占中央处理机时间的需要,都应进一步加强操作系统设计者和系统结构设计者之间的紧密配合。具体地说,应使系统结构设计者了解操作系统的进展,熟悉现在已在使用的资源、进程控制算法;而操作系统设计者则应了解系统结构的进展,尤其是硬件价格的下降和器件集成度的提高为系统结构功能的进一步扩大提供了很大的可能性,从而使操作系统设计者和系统结构设计者一起努力,进一步增强系统结构对操作系统的支持,并使尽可能多的操作系统功能转为由系统结构来直接实现。

如何增强系统结构的规整性(统一性、一致性、均匀性),也是改进系统结构使其优化于高级语言实现的另一个途径。规整的、一致的系统结构意味着没有或尽可能减少例外的情况和特殊的应用以及所有运算都能对称、均匀地在存贮(寄存)单元间进行。所谓对称、均匀地进行运算,就单元的使用来说,主要指的是在运算时,所有存贮(寄存)单元都可同等对待,不论是哪一个操作数或是运算结果都可无约束地存在任意单元。然而,目前机器都或多或少地不规整、不均匀。

就以 IBM370 来说,它的通用寄存器并不真正通用,例如有些指令用的操作数占用二个顺序的寄存器,但只能存在从偶数号寄存器开始的二个寄存器。又如有的指令只能使用第 1 号与第 2 号寄存器,有的指令(如字符串长传送指令)需用到四个指定的寄存器等等。这样,在编译时,对这些指令就需先要把它们要用到的寄存器腾出来,并把这些指令要用到的数据移入这些指定寄存器,这些都增加不是运算所必需的数据传送并使通用寄存器的优化管理复杂化。370 的指令系统也有不规整之处。例如,大多数面向全字(32 位)的指令(如取、存、加、减、乘等)都有对应的半字(16 位)指令,但除法指令却只有全字的,没有半字的。370 的有些特殊应用也给软件的设计带来不便,例如对“十进制变换到二进制”指令,其溢出却是以“定点除溢出”形式出现,从而还得由软件来判别当“定点除溢出”中断出现时,究竟是何种溢出。

PDP-11 指令系统也有类似的不规整性,不对称性,VAX-11/780 就注意于改进它,例

如,对减法指令 VAX-11/780 就既有 $A \leftarrow A - B$, 也有对称的 $B \leftarrow A - B$ 。指令系统的这种扩大,使得名称符号能易于对应到暂存位置,从而减少了名称冗余及相应的附加指令,这对于简化代码生成有好处。这种方法是基于所谓变换完整性概念,它所需增加的硬件量很小。

3.2-3 指令系统的发展

看来,今后的操作系统一般是不会用汇编语言编写,而是用高级语言编写。因此,指令系统的发展中,最主要的应是如何更好地面向高级语言的需要。然而,从本章一开始,我们就已多处讲过,高级语言与目前绝大多数的机器语言,在语义上是悬殊很大的。由于目前机器的系统结构基本上没有跳出 Von Neumann 型机器系统结构的框框,这就使得在把高级语言程序翻译成机器语言程序时,要执行大量的映象操作,即把高级语言的概念结构映象到与它差别悬殊的现有机器系统结构上。

既然一般机器的系统结构并不面向高级语言的需要,造成编译过程费时费事,那为什么这种状况一直没有得到根本改变呢?原因是多方面的。

从高级语言本身来看,如第一章 3.1-1 所述,是多样化的、不稳定的,至今还没有出现统一的高级语言,光常用的高级语言就有多种,而且还不断出现新的语言(如 PASCAL 语言就日益被大家所接受),甚至同一种语言也在不断地改进(如 FORTRAN 的逐步改进到 FORTRAN IV,直至最近的 FORTRAN 77),并且,各种语言的语法结构也差别较大。所有这些都使得系统结构的设计很难真正面向高级语言的实现。就从目前每台机器需配备多种高级语言来看,如果机器的系统结构是按优化于某种高级语言,例如优化于 FORTRAN 的实现来设计;就是说如果系统结构设计成其语义结构尽量接近于 FORTRAN 的,则对 FO-

RTRAN 的实现来讲,其效率必然是高的,然而对另一种语义结构与 FORTRAN 有较大差别的高级语言,其实现效率就可能比一般机器结构的还低。

下面我们以前面讲过的 FORTRAN DO 例子来讲,将其语义重画于图 2.24(a)。大家知道,不论 m_1 , m_2 , m_3 是何值,那“一串语句”肯定至少要执行一次(例如,对 $m_1 = 2$, $m_2 = 1$, $m_3 = 2$,即 DO n I = 2,1,2 这“一串语句”也会执行一次),这是因为 FORTRAN DO 的语义是先执行那“一串语句”后,才判“ $I > m_2$ ”否。

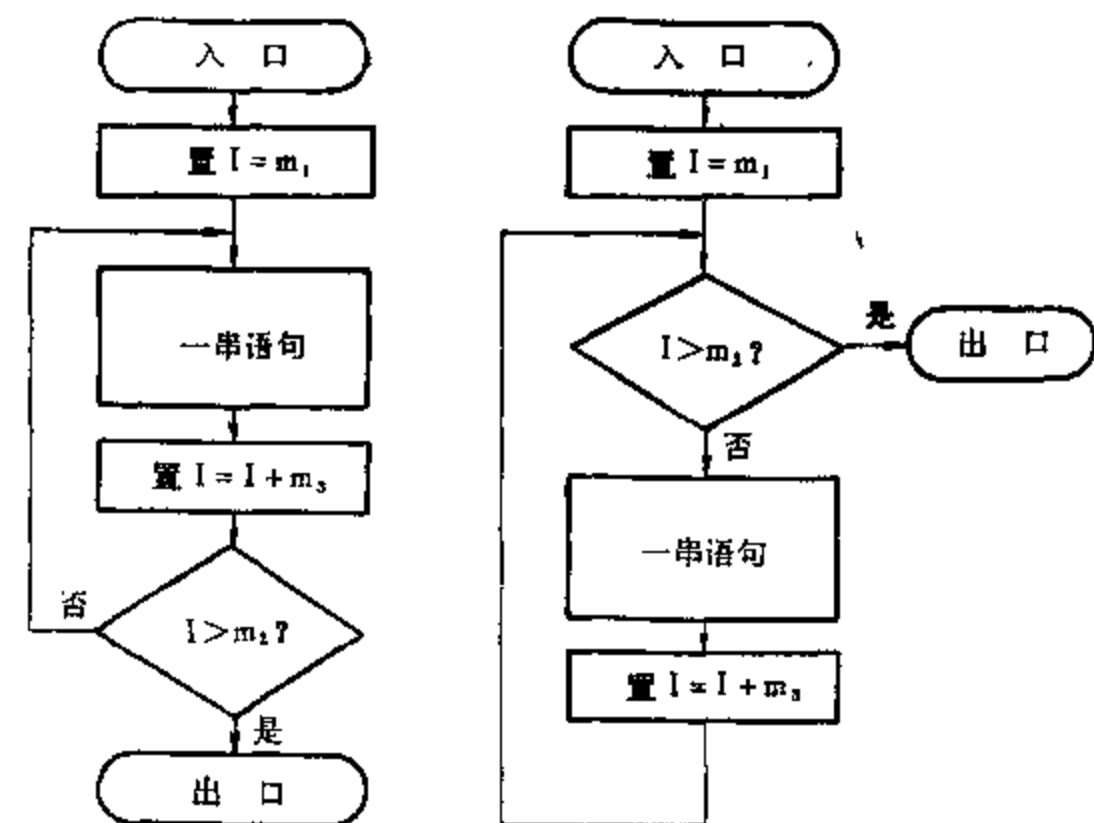


图 2.24 (a) FORTRAN DO 的语义
(b) ALGOL DO 的语义

如果我们使机器结构也有此语义,即机器也有这种语义的 DO 机器指令,则 FORTRAN DO 的实现就很方便直接。但是用这个 DO 机器指令去实现 ALGOL 的 DO 语句,那就麻烦了,因为 ALGOL DO 的语义如图 2.24(b)所示,对于

```

for  $l := m_1$  step  $m_3$  until  $m_2$  do
begin
  ⋮
end

```

是先判 $l > m_2$ 否，再执行 begin 到 end 之间的一串语句（若 m_1, m_2, m_3 取同上面 FORTRAN 一样的值，即 for $l := 2$ step 1 until 2 do 则这一串语句一次也不执行），这和 FORTRAN 的情况正好相反。因此，用与 FORTRAN DO 要求的语义相同或相近的 DO 机器指令来实现 ALGOL DO (COBOL PERFORM LOOP 也是如此，它的语义和 ALGOL DO 的一样，也是先判后执行) 比用通用的三条基本机器指令——取数（实现 $l = m_1$ ）、相加（实现 $l = l + m_3$ ）、大于转移（判 $l > m_2$ 否）——来实现还要麻烦和低效。而且，由图 2.24(a) 可见，用同样这三条机器指令也可实现 FORTRAN DO 要求的语义。因而，人们往往宁愿用上述基本指令实现 FORTRAN DO 和 ALGOL DO，使得对各种常用高级语言都具有相近的效率，而不去采用只对其中一种高级语言高效，却对其它高级语言的实现是低效的机器指令。

由对实现 DO 的分析可以看出，我们不能只从一种高级语言的需要来改进机器语言的语义结构。因为这种做法，从整体来看可能是害多利少。有人认为，各种高级语言之间以及它们与一般机器（即 Von Neumann 型机器）系统结构之间，在语义上的差别可用图 2.25 表示。图中，用各种语言与 Von Neumann 型结构点间的“路长”表示它们的语义差别，路径愈长的其实现效率愈低。看出，若系统结构点过份接近于某种语言，则这种系统结构与其它语言间的路径可能比距离 Von Neumann 型结构点还要远，即它们之间的语义差别还要大，实现效率还要低。

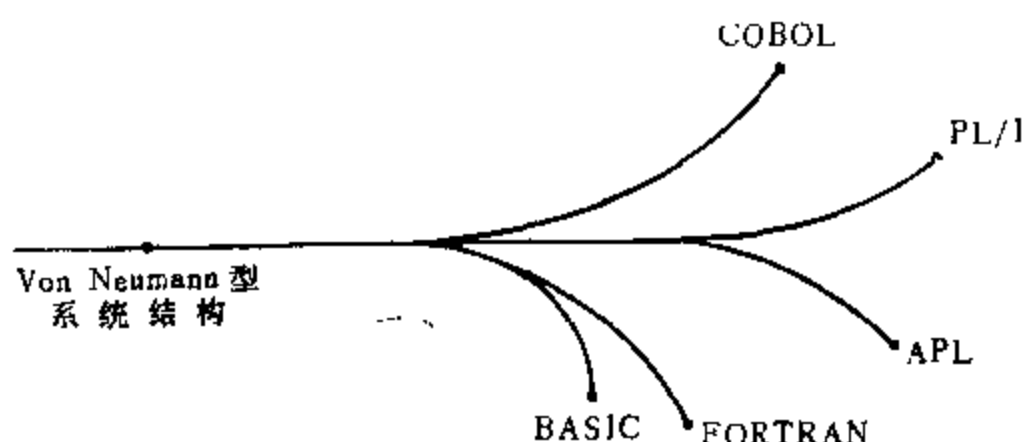


图 2.25 各种语言与 Von Neumann 型系统结构的语义差别图

因为这种做法，从整体来看可能是害多利少。有人认为，各种高级语言之间以及它们与一般机器（即 Von Neumann 型机器）系统结构之间，在语义上的差别可用图 2.25 表示。图中，用各种语言与 Von Neumann 型结构点间的“路长”表示它们的语义差别，路径愈长的其实现效率愈低。看出，若系统结构点过份接近于某种语言，则这种系统结构与其它语言间的路径可能比距离 Von Neumann 型结构点还要远，即它们之间的语义差别还要大，实现效率还要低。

接近于某种语言，则这种系统结构与其它语言间的路径可能比距离 Von Neumann 型结构点还要远，即它们之间的语义差别还要大，实现效率还要低。

这样，人们只得把系统结构设计成比较通用和基本，使各种常用语言与它在语义上的差别程度尽可能相近；就是说，使各种常用语言与系统结构点间的“路长”尽可能相同，那怕对每一种语言来讲，都远不是优化的。而且，在这种通用、基本的结构上，新的高级语言总是可以经编译来实现；而如果把结构设计成过份接近于某种高级语言，则对于新的或某种高级语言甚至可能会出现无法实现的问题。目前，一般机器的系统结构，其指令系统就是这种通用和基本的。当然，结构是通用和基本的也便于应付各种语言的不断改进，因为在它的基础上总是可以实现，为此只需相应的少许修改编译软件即可。这就是至今一般机器的系统结构设计难于真正面向高级语言实现的主要原因之一。

其次是长期以来，计算机系统的设计原则是使硬件尽量基本、简单，而复杂、麻烦的事尽可能靠软件来解决，这就使得硬件的设计多数是面向怎样使硬件本身的实现方便和高效，而较少考虑如何更便于软件的设计。此外一个原因是硬件设计和软件设计的过份分家，缺少

相互间的渗透。

当然，绝不是说在保持通用性的前提下，从程序设计者看的系统结构只能是 Von Neumann 型最初那种样子。从图 2.25 也可看出，我们是有可能把系统结构点往右移，使得它与各种语言间的路长都得到缩短。还是以上面所讲 DO LOOP 的实现为例，通过设置上一小节讲述的“相加比较与转移”复合指令（IBM370 称为“大于转移”指令），它对加快 FORTRAN DO 及 ALGOL DO（包括 COBOL LOOP）的实现都有好处。这点，只需把图 2.24，按照各自的 DO 语义变成图 2.26，就可看清。增加这条复合指令，减少了机器系统结构与 FORTRAN 以及 ALGOL 等的语义差别，几乎以同等程度缩短了图 2.25 中系统结构点与各语言间的路长。如前所述，这种改进当然是可取的。前面讲过，对于微程序控制机

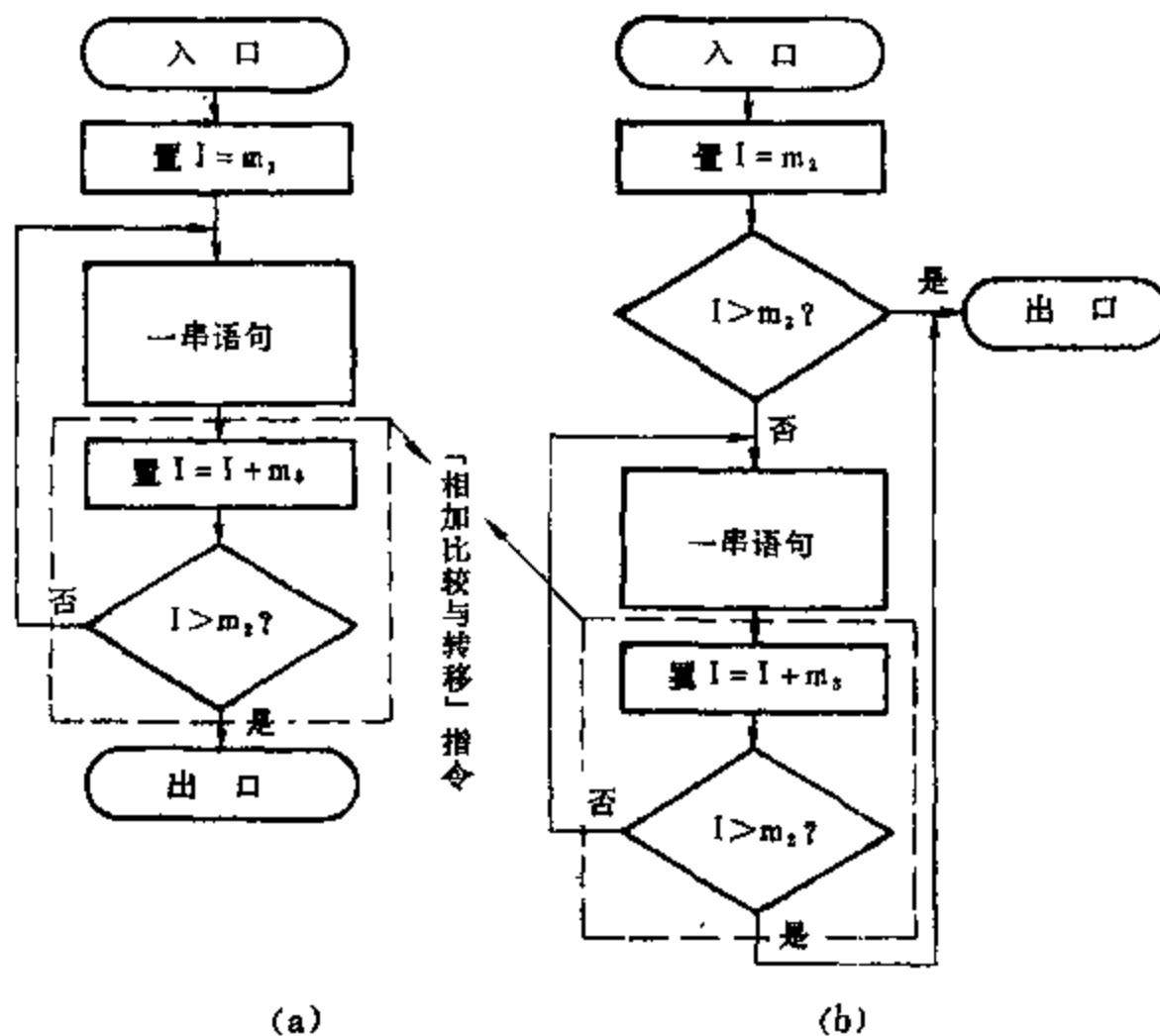


图 2.26 用“相加比较与转移”指令加快 DO 的实现

器，增添这类复合指令往往只需少许增加控存容量，但却使得各种语言的编译程序都得到简化，从整体看是合算的。当然，复合指令的增添一定要考虑到它的利用率究竟有多少。要注意到，由编译形成的目的程序中，远不是所有机器指令都能用到，一般计算机系统只用到其中的百分之二十到五十，而且经常用的往往只达百分之十（近年来，由于系统结构与软件设计的配合加强，机器指令的利用率有了提高，VAX-11 的 FORTRAN 编译所形成的目的程序中，这个比值提高到百分之七十五）。因此，不论所增加的复合指令其语义如何接近于高级语言的，若其利用率很低，这种增加就没有多大实际意义。

虽然从六十年代初使用高级语言以来，在 Von Neumann 型结构的基础上，围绕怎样方便于高级语言的实现，是有了包括上面和前一小节讲的不少改进。然而，二十年来指令系统的变化并不显著，难题之一就是我们在前面讲过的，各种高级语言所要求的优化指令系统并不相同，甚至差别较大。这个难题如何解决呢？可以想到的办法之一是使机器具有多种系统结构，每种分别面向某种高级语言，且这多种系统结构能够动态地进行切换，使得在运行各

种高级语言时，分别有面向它的系统结构与之对应。随着六十年代微程序技术的发展，特别是可读写控存的采用，就给这种动态结构机器的实现提供了可能（在第一章 § 2.2-3“微程序”中已提到过这种思路）。于是，在 1972 年 Burroughs 公司就按此想法制成了 B-1700 机器。它的每种高级语言，如 BASIC, FORTRAN, COBOL 等，各对应一个由微程序解释的面向这种语言的系统结构，因为 COBOL 与 RPG 的语义相近，所以这二个语言共用一种系统结构。由于它的操作系统是用称谓 SDL（与 ALGOL 及 PL/I 相似）的语言进行编写，为了提高操作系统的执行效率，还设计了面向 SDL 的系统结构。每种系统结构各有其自己的指令系统与数据表示格式。这样，B-1700 就具有多种系统结构，如图 2.27 所示。当操作系统判定要执行 COBOL 程序时，它首先要切换现行微程序，使机器变为具有面向 COBOL 的系统结构，而后执行 COBOL 编译程序，把 COBOL 源程序翻译成面向 COBOL 的机器语言程序，它的每条指令是经微程序解释实现。当执行到需要调用操作系统时（如要处理 I/O 中断时），又先切换微程序，使机器具有面向 SDL 的系统结构，而后操作系统才运行。每种高级语言所对应的微程序约需 2000 条 16 位长的微指令，当然不需要把全部语言所对应的微程序都同时存在控存内，而可以把大部分存在主存内，只当要用到那个高级语言时，才把

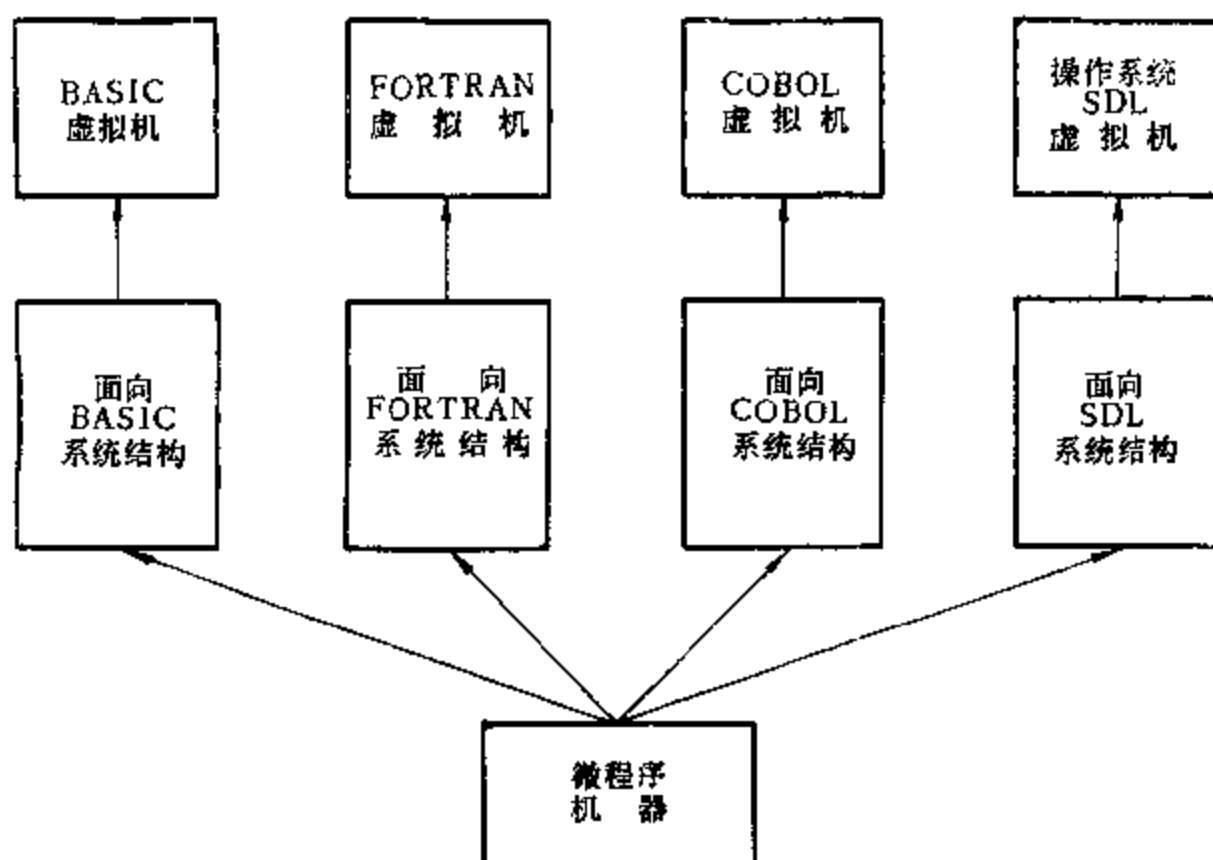


图 2-27 B-1700 具有多种系统结构

对应于它的微程序调进控存内。在 B-1700 之后的改进型 B-1800（76 年问世）引用了第五章要讲的高速缓冲（Cache）存贮器的概念，使微程序能根据需要由主存调进微程序 Cache 存贮器。

应该说，B-1700 是企图突破传统 Von Neumann 型机器的一个有意义的尝试。它不断进行改进，B-1800 又进一步改进为 B-1900（79 年问世）。在 B-1700 之后，于七十年代中间问世的 IBM-5100 小型机有点和它类似，也是通过微程序技术使得在运行 APL 程序时，机器具有和 IBM370 相近的系统结构，而在运行 BASIC 程序时，却具有 IBM System/3 的系统结构。不过，IBM-5100 不是通过编译实现高级语言，而是通过解释去实现。

虽然 B-1700 的这种设计思路从 72 年问世以来，还没有更多的机器采用它，主要原因

恐怕是在于它对实现高级语言的效率还没有比通常 Von Neumann 型结构有明显的提高。然而,这种使系统结构设计优化于多种高级语言实现的出发点是很可取的。而且,这种多结构动态切换还可以看成是自适应(或曰自寻优)计算机系统很粗糙的雏型,因为机器的结构能够根据不同的使用状况(不同的高级语言),通过微程序的切换自动地去适应它(即使之具有面向使用状况的系统结构)。这种动态结构比之于 Von Neumann 型固定结构是前进了一步。自适应计算机系统的研究是吸引人的,例如,有人研究了面向题目的自寻优计算机系统,它的编译系统对源程序进行分析,找出哪种语句或是哪种类型的一段语句是用的最频繁的,并判定用一条机器指令去执行一条或一段语句在效率上是否确是带来明显的好处。而后,就按此确定面向这个源程序的优化机器指令系统,并把对应这个指令系统的微程序形成好送入控存。编译程序于是就按此优化了的指令系统去形成目的程序。这是很有意思的想法。当然,应该采用什么样的优化算法(上面只是一例),如何去实现它,如何评价它的实际效果等等,还有大量的问题需要解决。

B-1700 的系统结构只是面向高级语言,高级语言源程序还是要经编译形成机器语言目的程序才能执行。就是说,前面图 2.2 所示的分析与综合这二步都仍然执行,但由于系统结构的语义是面向源程序所用的高级语言,因而其实现效率会提高。

在上述分析的基础上,能否使机器的系统结构更进一步面向于高级语言的实现呢?应该说,我们在前面提到过,而且在下一节要详细讲述的堆栈机器,在设计时是很好地考虑了面向高级语言实现的需要。然而,上述所讲的种种系统结构,其高级语言的实现仍然是按图 2.2 所示,编译过程仍是经软件实现。那么,是否有可能突破这个框框?突破的思路可以有哪一些呢?

一种思路是能否由硬件或固件去执行中间语言程序。参看图 2.2,就是说,使得机器指令系统的语义与中间语言的相符。这种思路是从六十年代末就已有有人提出并进行研究,不过,很难设计出一种对各种高级语言都是优化的中间语言,至今这种思路进展不大。

另一种思路就是能否使得机器语言升高到具有和高级语言一样的或非常相近的词法、语法和语义结构,这就是所谓高级语言机器,高级语言机器一般没有编译软件。

一种高级语言机器有着与该高级语言一一对应的机器语言,是用汇编的方法(可以用软件实现汇编,也可以是用硬件实现)把高级语言源程序翻译成机器语言程序。七十年代初制成的 SYMBOL 试验性机器就是属于这种类型,它的指令系统是与此机器系统所用的程序设计语言 SPL 一一对应。其实,它的机器语言程序仍然是用编译方法实现,但编译功能是用硬件实现,而且程序定位,虚拟存贮系统等等以至对终端命令的解释都全是用硬件实现。这台机器的制成充分说明硬件和软件是等效的,我们是可能采用硬件去实现原由软件实现的功能。然而,由于这台机器几乎没有软件,而且又没采用微程序技术,因而在制成后其结构就定死了,可是这种结构所能支持的 SPL 语言并不是大家所熟悉的常用高级语言。SYMBOL 机器虽只生产了一台,但是,这种不用软件的大胆尝试还是有意义的,它的研制和运行分析为进一步研制高级语言机器提供了经验和教训。

另一种高级语言机器本身没有机器语言,而是直接由硬件和固件对高级语言源程序的语句逐条进行解释以执行它。近年来对这种形式的高级语言机器的兴趣有所增加,它既没有编译,也不用汇编,而完全用或主要用解释方法实现高级语言。由于是逐条解释,当发现有程序设计的错误时,错误现场容易保存,从而也容易排除;另外,解释方法对实现交互式语言

有利。人们称这种高级语言机器具有直接执行的系统结构。

应该说，高级语言机器这种系统结构适应于集成度迅速提高、硬件价格不断下降这个发展趋势，是符合于今后要求减少软件复杂性，增加系统中硬件比例的要求，值得我们去研究。然而，这个思路是早在六十年代初就已有入提出，虽然之后不断有人进行研究，也制成过包括 SYMBOL 在内的个别试验性机器；可是，十几年过去了，至今还没有一种高级语言机器已在出售，或已在成批生产。原因何在呢？

原因是多方面的。但看来最主要的原因在于已有的高级语言机器，甚至包括已公布的高级语言机器设计方案，还没有显示出它确是比按现有机器系统结构形成的计算机系统，有更好的性能价格比。

首先，目前的计算机系统具有包括系统程序语言在内的多种高级语言。前面说过，这是由于至今还没有一种语言能满足各个方面的需要之故。然而，高级语言机器适于只面向单种高级语言（或是语言结构很相近的少数几种高级语言），因为只有这样，才有可能使得它能比目前计算机系统更高效地实现该种高级语言。固然，随着集成度的迅速提高，是有可能实现单片的各种高级语言机器；从而对于需配置多种高级语言的计算机系统，可以通过在机器内装设应对于各种高级语言的各种片子来解决。但是，要真正做到这点，还有很多难题没有解决。例如，直接执行的高级语言机器，其实现效率在很大程度上取决于该种高级语言是否适合于用解释实现。况且，当前常用的各种高级语言并不都是能有效地用解释方法实现，例如，BASIC、APL 等语言是适合于对逐条语句用解释方法实现，但 FORTRAN、COBOL 语言用解释方法实现其效率会下降，而 ALGOL 和 PASCAL 语言则是要求在执行前要对整个程序进行预处理以解决各个语句间的上、下联系。就是对于可用解释实现的语言，也不是所有语句都可用逐条解释实现。例如对“PROCEDURE CALL”语句，究竟应转移到哪里，只通过逐条解释是难于实现的。

其次，还应认识到，计算机系统的性能好坏不只是反映在高级语言的实现效率上，几乎在同等程度上还反映在所配操作系统的性能及其实现效率上。目前，操作系统几乎占用了计算机系统的三分之一资源。因此，如果高级语言机器只支持用户语言的高效率实现，而对操作系统语言没有支持或支持较差；没能使得操作系统中，诸如进程管理、存贮管理、文件存贮、外围接口、文本处理等具有比现有计算机系统更好的性能和更高的实现效率，那用户是不会对这种机器系统有很大兴趣的。

还有，就是在提高高级语言的实现效率上，高级语言机器过去主要着眼于如何提高“代码生成”（参看图 2.2）的效率。然而，这只是编译过程的一部分；编译还包括语法分析、语义分析和词法分析。不只是能发觉语法、语义和词法上的错误，还尽可能指明错误所在，已日益成为当今编译系统的一个重要功能。只有高级语言机器也能做到这点，并能把发觉出的错误返回映象到源程序，即能指明是哪条语句错了，用户才能认为高级语言机器确是比现有机器系统更便于使用。充分应用和发展前面讲过的自定义数据表示和尽量减少语义差距，高级语言机器是有可能做到能指出错误语句的所在。

最后，虽然说软件和硬件在逻辑功能上是等效的，而且软件的硬化是今后的必然发展趋势；然而，简单地将现有软件的算法全盘改由硬件实现，其效果往往并不好。SYMBOL 机器就是一个例子，它的硬化了的编译系统和操作系统用了好几年时间才调出，可是其效能并不比现有机器系统的强。看来，把编译和操作系统中的基本操作，如上一节讲的那些，改由

硬化实现是可取和合理的，而对于复杂算法的硬化则要很谨慎。愈是复杂的算法，其设计费用愈高，其实现费用也会比软件实现的大，而其效率并不一定会比软件实现的高；况且，并不是任意复杂的算法都能硬化实现。还应注意到，复杂算法的硬化实现肯定不能一次设计、调试成功；因此，所设计的硬件系统必须可以灵活修改，便于调试。这样，固件（微程序）实现就要比硬件实现合理。

总之，高级语言机器并不是非得一点软件也不能要，而是应着眼于如何发挥大规模和超大规模集成电路的作用，改变现有的软、硬件功能分配，增加由硬件实现的比例，落实于提高整个计算机系统的性能价格比。

§ 4 堆栈机器的结构特点

我们之所以单独设置一节来讲述堆栈机器的系统结构，不只是因为它与通用寄存器型机器的系统结构有比较大的差别，还由于它的设计是注意于改进从程序设计者看的系统结构，为诸如高级语言程序的编译和子程序的调用等提供更好的支持。我们曾经多次讲过，革新、改进从程序设计者看的系统结构将是八十年代系统结构的最主要进展。因此，具体的了解堆栈型机器的系统结构特点是有必要的。

在五十年代末期就提出堆栈数据结构，并得到了应用，开始时它当然是在一般 Von Neumann 型机器上经软件实现。但是，当时就已有人致力于研究面向堆栈的机器。到了六十年代初，开始具体的设计。于是，美国的 B-5500 和英国的 KDF-9 堆栈型机器在 1963 年问世了。之后，堆栈型机器与以 IBM360（也是 63、64 年开始问世）为代表的通用寄存器型机器一样，也得到了发展。Burroughs 公司继 B-5500 之后，于六十、七十年代推出了 B-6700、B-7600，B-6700 又发展成 B-6900。还有，Hewlett-Packard 公司于七十年代初推出了 HP-3000 堆栈小型机，它吸收了大型堆栈机器的成功之处，不断改进，至今还在生产。另外，从六十年代末期以后制成的小型机，有些既具有通用寄存器，也具有对堆栈的某些支持，PDP-11 就是突出的例子。NOVA 机发展成 Eclipse 机（75 年问世），加进某些面向堆栈的机器指令和结构是其重要特征之一。微处理器更是如此，最初的几个主要型号，如 Motorola^a 6800、Intel 8080 都具有某些堆栈功能，自然，其后继者及与这二者兼容的其它微处理器也是如此。当然，不论是 PDP-11、Eclipse 还是上述那些微处理器，都不能说是堆栈型机器，这点我们下面会讲到。

本节首先简要地回顾堆栈数据结构及其实现；而后结合算术表达式的求解和子程序的调用与链接，讲述堆栈型机器的基本特征；最后，简要地对堆栈型机器与通用寄存器型机器进行比较。

4.1 堆栈数据结构及其实现

大家从“数据结构”课已知，堆栈数据结构是这样一种有序表，它的插入和删去只能从称为栈顶的一端进行，表的另一端称为栈底。从栈顶插入一个元素

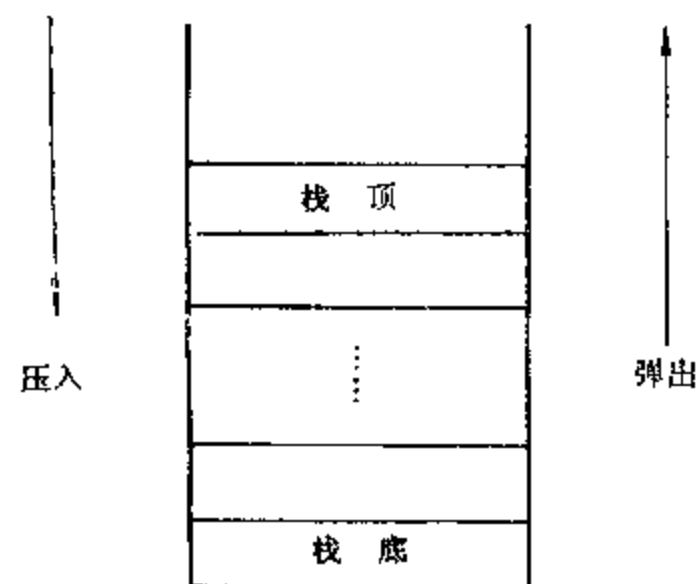


图 2.28 堆栈的先进后出有序表

称为压入（PUSH），从栈顶删去一个元素称为弹出（POP），如图 2.28 所示。由于压入

和弹出都只能从栈顶这一端进行，所以这种有序表是后进先出（LIFO）的。从原理上讲，堆栈也可从栈底起始任意扩大。

这种数据结构当然要以某种方式在机器内实现。在对堆栈没有任何支持的一般 Von Neumann 型机器上的实现，如图 2.29 (a) 所示。

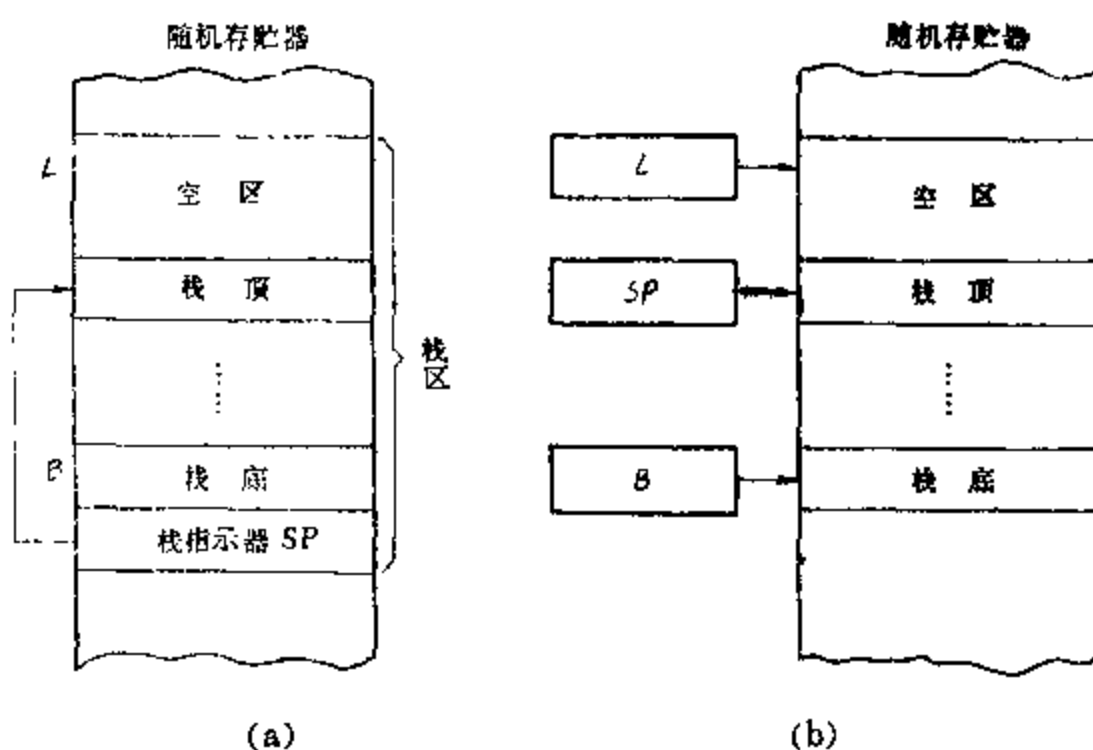


图 2.29 堆栈在随机存储器中的实现

在主存中为此堆栈划定一个栈区，栈顶单元的地址存在栈指示器 SP (Stack Pointer) 单元内。若堆栈是向上生长（即栈顶单元的地址值比栈底的大），则

PUSH X	$SP \leftarrow SP + 1$
	$M(SP) \leftarrow M(X)$
POP X	$M(X) \leftarrow M(SP)$
	$SP \leftarrow SP - 1$

本节采用第八章要讲的 CDL 硬件描述语言符号， $M(X)$ 为主存内 X 地址单元的内容。若堆栈是向下生长（如 PDP-11 和某些微处理器那样），则

PUSH X	$SP \leftarrow SP - 1$
	$M(SP) \leftarrow M(X)$
POP X	$M(X) \leftarrow M(SP)$
	$SP \leftarrow SP + 1$

选择向上或是向下生长是任意的。

显然，堆栈不能随意生长。对于堆栈是向上生长的，在压入时不能出现 $SP > L$ (L 称为栈区上界)，否则它会侵犯别的程序块，称为出现“上溢”；同理，在弹出时不能出现 $SP < B$ (B 称为栈区下界)，否则会把 SP 或是别的程序块内容当作堆栈元素，称为出现“下溢”。当然，上、下溢的判别可以由软件实现，但是要费时间。

可见，为实现 $SP \leftarrow SP \pm 1$ 操作，需访存二次（取 SP 、存 $SP \pm 1$ ），这会使堆栈的压入和弹出过份费时间。因此，为了加快这一操作，给堆栈的实现提供支持，有些机器（如图 2.29 (b) 所示）就设置了 SP 寄存器，即消除了二次访存，也使“ ± 1 ”有可能直接在寄存器进行，而不必经运算器实现。如果再增设 L 、 B 寄存器，并使上、下溢的判别由硬件（或固

件) 实现(堆栈机器都有这种功能), 还会进一步加快压入、弹出和其它堆栈操作。

本来按堆栈的原来定义是只能访问栈顶, 然而, 在实际应用中, 若是也能访问到栈内的某些元素, 则会给堆栈的使用带来方便。所以在堆栈机器中, 往往又设置一个或多个指示器, 指示栈底和栈顶间的某些位置, 并使机器能访问到相对于这些指示器的单元, 这点在 § 4.3 还要详细讲。

由于堆栈设置在主存内, 对它的使用终究要经访存操作, 所以, 速度就快不了。尽管可以用多个并行寄存器或多个移位寄存器实现堆栈, 以提高速度, 如图 2.30 所示(由于是寄存器实现, 所有元素可同时上、下移。在这里, 栈顶位置不动, 压入时栈底位置下移, 弹出时栈底位置上移); 但是, 用这种方法实现整个堆栈, 其设备量会过大(K 值可能超过 1000,

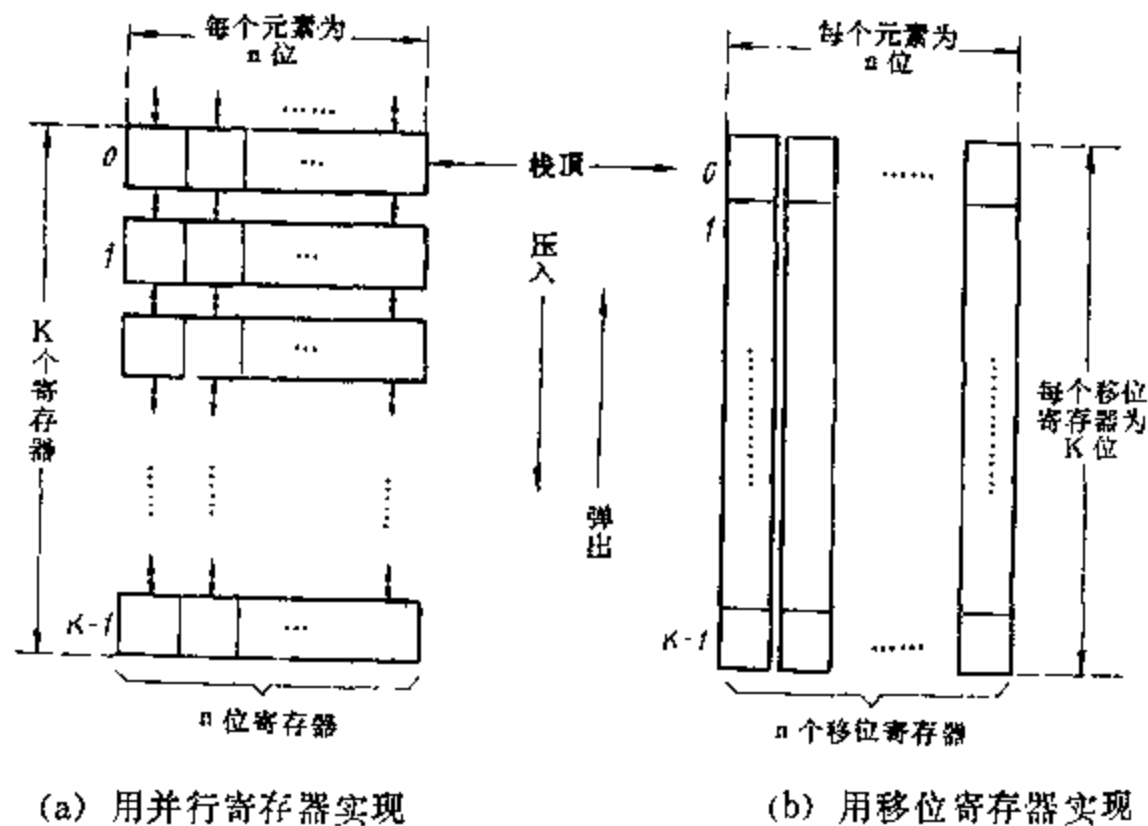


图 2.30 堆栈的寄存器实现

K 为栈区深度)。所以, 目前的堆栈机器是采用寄存器实现和主存实现相结合的办法, 这点在下节再讲。

4.2 算术表达式的求解

本节讲述为什么堆栈机器便于实现复杂算术表达式的求解。为此, 先简要地回顾“逆波兰表达式”, 而后简述堆栈机器的算术运算型指令及提高指令速度的一些措施。

4.2-1 逆波兰表示式

大家知道, 运算型机器指令一般只能对二个操作数进行一种算术运算(如相加); 但是, 高级语言却是可用一条赋值语句求复杂算术表达式的解。那么, 如何由一个算术表达式, 例如 $A/B + C * (D + E)$, 形成解此表达式的机器语言程序呢? 当前的编译程序几乎都是先把算术表达式变换成逆波兰表示式, 而后再由它形成机器语言程序。

有多种算法能把算术表达式变换成逆波兰表示式。其中, “后序走树法”是常用的一种。

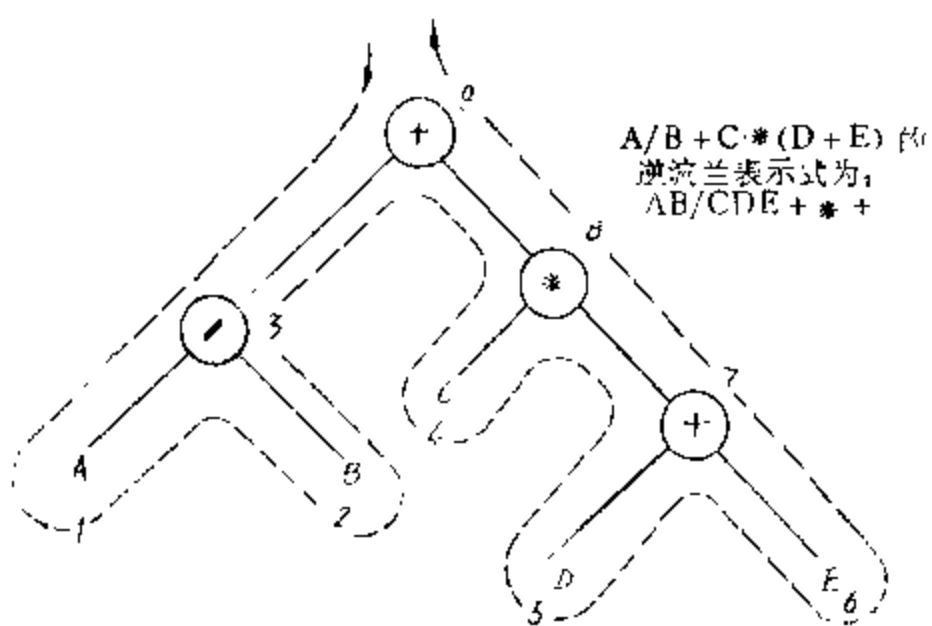


图 2.31 $A/B + C*(D+E)$ 的树形表示

对它, $A/B + C*(D+E)$ 表达式用树形表示如图 2.31 所示, 结点为运算符, 叶为操作数。后序走树法的走法如图上虚线所示, 按先后次序写下所经过的每个叶及其左、右子树都已走过的结点, 这样, 按图上所标号数写下变量和运算符, 就得到其逆波兰表示式, 它不需要括号, 只需从左到右执行就能得到结果。由逆波兰表示式形成堆栈型机器的机器语言程序, 要比形成通用寄存器型机器的方便和直接得多, 这点下面就要讲到。

4.2-2 堆栈机器的运算型指令和栈顶寄存器

堆栈机器运算型指令的运算结果恒保留在堆栈内。操作数中的一个一般是在栈顶, 另一个操作数或是在次栈顶 (即栈顶的下一个单元), 或是在主存内的某个单元, 或是指令内的直接操作数。下面结合 HP-3000 的指令系统来讲。

HP-3000 把操作数在栈顶、次栈顶的指令称为堆栈指令, 例如有:

ADD	$M(SP-1) \leftarrow M(SP-1) + M(SP)$
	$SP \leftarrow SP - 1$
MUL	$M(SP-1) \leftarrow M(SP-1) * M(SP)$
	$SP \leftarrow SP - 1$
DIV	$M(SP-1) \leftarrow M(SP-1) / M(SP)$
	$M(SP) \leftarrow \text{余数}$
DEL	删去 (清除) $M(SP)$
	$SP \leftarrow SP - 1$

显然, 这类指令不需要有操作数地址码 (因此有人就称之为零地址指令), 从而指令可缩短。但若指令不是可变长的, 则程序的总位数并不能因此有所减少, 而仅仅使堆栈指令字出现信息冗余量。HP-3000 的指令是定长 (16 位), 主操作码是 4 位、地址码是 12 位。对于堆栈指令, 它采用扩展操作码法, 如图 2.32 所示: 当主操作码为 “0000” 时, 表明为堆

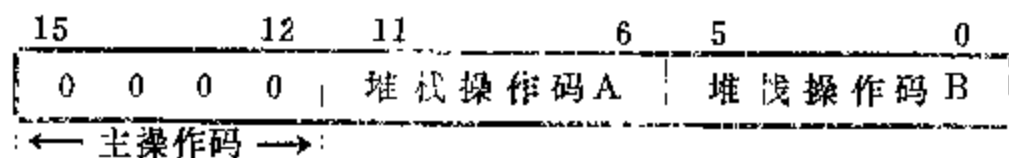


图 2.32 HP-3000 的“堆栈指令”格式

栈指令, 执行何种操作由堆栈操作码指明。为表示包括上述 ADD, DIV、DEL 等几十种堆栈指令, 指令的长度本只需:

4 位 (主操作码) + 6 位 (堆栈操作码) 共 10 位就够了。为了利用冗余的 6 位, HP-3000 再设置一个堆栈操作码。这样, 由一条堆栈指令可控制二个堆栈操作 (先执行 A 的, 再执行 B 的)。例如, 若 A、B 操作码为:

ADD ADD

则可执行三个元素的连加，而若是

DIV DEL

则删去余数，栈顶为商。这种安排是增加堆栈机器定长指令字信息量的好办法。

HP-3000 的另一类运算型指令，有一个操作数是由地址指明，称为访存指令，例如有：

LOAD X (同上述 PUSH)	$SP \leftarrow SP + 1$ $M(SP) \leftarrow M(X)$
STOR X (同上述 POP)	$M(X) \leftarrow M(SP)$ $SP \leftarrow SP - 1$
ADDM X	$M(SP) \leftarrow M(SP) + M(X)$
MULM X	$M(SP) \leftarrow M(SP) * M(X)$

有了这些运算型指令，就能直接方便地由逆波兰表示式形成机器语言程序。以图2.31的式子为例，编译程序可直接按逆波兰表示式 $AB/CDE + * +$ 从左到右的顺序形成如下机器语言程序：

LOAD A	$M(SP) \leftarrow A$
LOAD B	$M(SP) \leftarrow B, M(SP-1) \leftarrow A$
DIV DEL	$M(SP) \leftarrow A/B$
LOAD C	$M(SP) \leftarrow C, M(SP-1) \leftarrow A/B$
LOAD D	$M(SP) \leftarrow D, M(SP-1) \leftarrow C$ $M(SP-2) \leftarrow A/B$
ADDM E	$M(SP) \leftarrow D + E, M(SP-1) \leftarrow C$ $M(SP-2) \leftarrow A/B$
MUL	$M(SP) \leftarrow C * (D + E)$ $M(SP-1) \leftarrow A/B$
ADD	$M(SP) \leftarrow A/B + C * (D + E)$

显然，最后二条指令可用上述具有双堆栈操作码的一条堆栈指令

MUL ADD

替代。看出，对堆栈机器，存中间结果的工作单元的安排要方便得多。

可见，用堆栈结构实现逆波兰表示式是方便和直接的，因此我们说，堆栈机器的设计是面向高级语言，面向“代码生成”的实现。当然，不能只看到“代码生成”的便于实现，还需分析堆栈机器指令的实现速度。

上面讲的是包括栈顶在内的整个堆栈都存在主存内，即由栈顶、次栈顶取操作数，以及把每条指令的运算结果压入栈顶都需经访存操作。这当然会使堆栈机器执行指令速度比通用寄存器型机器的慢得多，而前面讲过，要使整个堆栈都由寄存器构成，代价又过高。然而，由堆栈指令的介绍可看出，执行堆栈指令时，要用到的只是栈顶和次栈顶单元，这样，若使它们由寄存器构成，就能大大减少访存次数，而所需增加的硬件又很少。不过，由上述具体程序例子看出，中间结果 (A/B) 会压入 $SP-2$ 单元，因此光是 SP 和 $SP-1$ 二个单元由寄存器构成还不够，而是希望能有更多的单元由寄存器构成。所以，堆栈机器至少是栈区的头

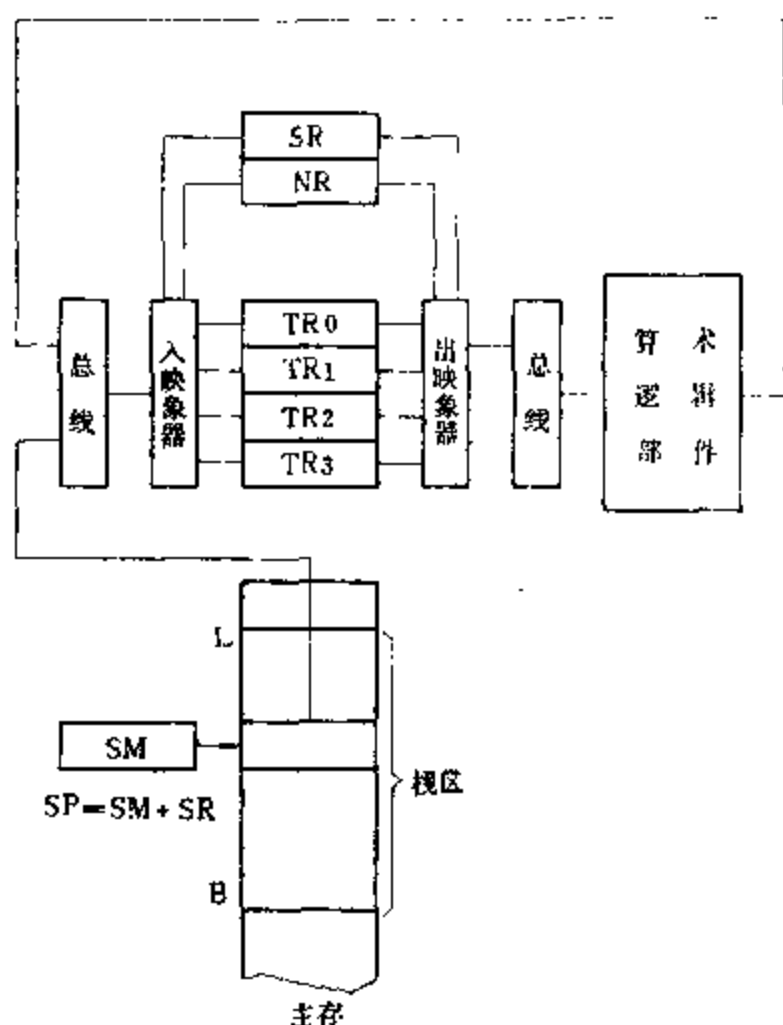
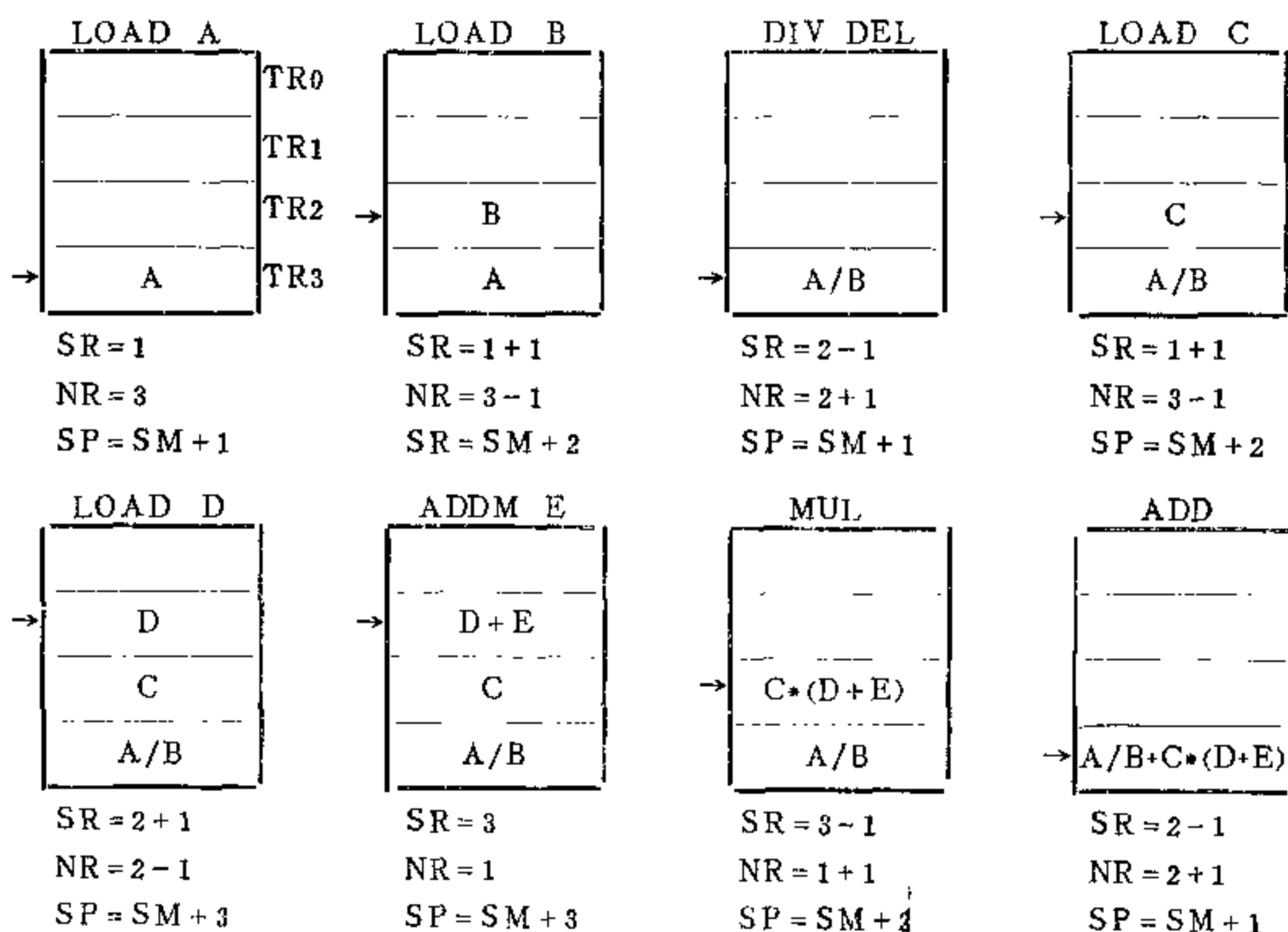


图 2.33 HP 3000 的运算堆栈结构

-1 动作的，则执行 $SR-1$ ， $NR+1$ 。图 2.34 是执行上述求 $AB/CDE + * +$ 的机器语言程



“→” 指向栈顶

图 2.34 HP-3000 栈顶寄存器的使用例子

序时, 栈顶寄存器内容和 SR、NR、SP 的变化。

看出, 对堆栈的访问可在栈顶寄存器进行, 大大加快了指令的执行。但是, 在 $SR=4$, 即栈顶寄存器已被对应于 SP 到 SP-3 的四个元素占满, 而又要执行 SP+1 时, 则需先把 SP-3 元素压入主存内的堆栈, 对应地 SM 要加“1”, SR 要减“1”; 而后, 才能执行对应 SP+1 所需的操作。同理, 在 $SR=0$, 而需对栈顶进行运算时, 则需先把栈顶由主存弹入栈顶寄存器, 对应地要 SR 加“1”, SM 减“1”; 而后才能执行对栈顶的运算。还有, 在 $SR=1$, 但指令的运算需用到次栈顶时, 也需先把次栈顶由主存弹入栈顶寄存器。这些判定和操作都由微程序实现。

就这个例子来讲, 栈顶寄存器有三个就够了。然而, 在实际算题时, 在多道程序环境中,

栈顶寄存器应取多少个为宜呢? 当然是个数愈多, 访存概率愈低; 但是, 在超过一定个数后, 个数的继续增大往往不会使访存概率再下降多少。为了判定 HP-3000 选取四个栈顶寄存器是否合适, 在实际机器上经微程序仿真, 使之似乎栈顶寄存器个数可变 (除了真有四个寄存器之外, 还用主存单元作为伪栈顶寄存器)。而后, 对不同的寄存器个数(仿真的)记录栈顶寄存器与在主存内的堆栈部分之间交换信息的次数。它包括压入和弹出二种, 这里讲的压入指的是栈顶寄存器装不下, 需把原在寄存器的元素压入主存; 弹出指的是 CPU 所需用的堆栈元素不在寄存器, 需由主存弹入寄存器。图 2.35 是在运行多道程序时测得的 每条指令的压入、弹出平均次数与寄存器个数的关系曲线; 这里的数值不只是对应于上述运算型指令的, 还包括子程序调用指令。

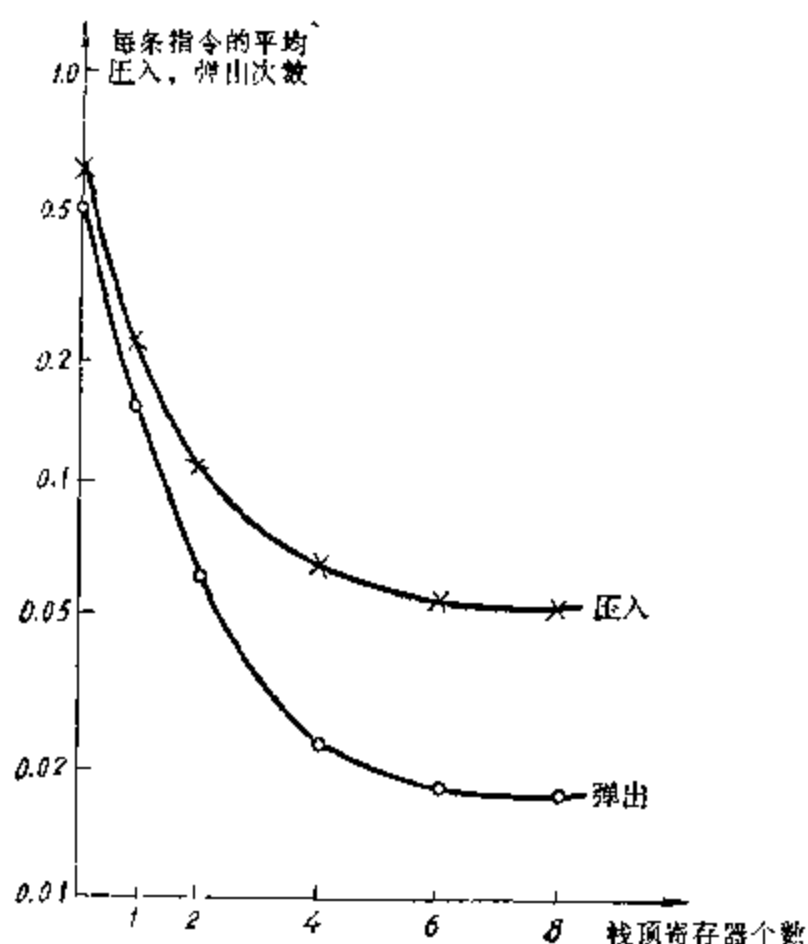


图 2.35 压入、弹出次数与寄存器个数的关系曲线

看出, 对 HP-3000, 寄存器个数取为 4 是接近于优化值, 因为个数的继续增加并不会使压入、弹出次数有明显的减少。然而, 这个结果与 HP-3000 指令系统的设计以及操作系统的设计都是从只有 4 个栈顶寄存器出发有密切关系。不过, 对于象 HP-3000 这一类堆栈机器, 只需个数不多的栈顶寄存器就能大大减少主存内堆栈的压入和弹出, 这个结论是肯定的。

4.3 程序的调用与链接

上一节讲了堆栈机器在解算术表达式方面的优点, 这一节则是讲述堆栈机器为何对实现子程序的调用与链接特别有利。先讲讲堆栈结构是实现程序调用的有效工具; 而后, 结合 HP-3000 具体讲述程序调用指令和返回指令。

4.3-1 堆栈结构是实现程序调用的有效工具

大家知道, 程序的模块化和子程序概念是程序设计的重要技术, 因此, 高级语言的

CALL 和 RETURN 语句是经常使用的。然而,若这些语句的执行很费时间,那子程序技术的好处就难于得到发挥。所以,如何为 CALL 和 RETURN 的实现提供更好的支持,是改进系统结构的一个方面。

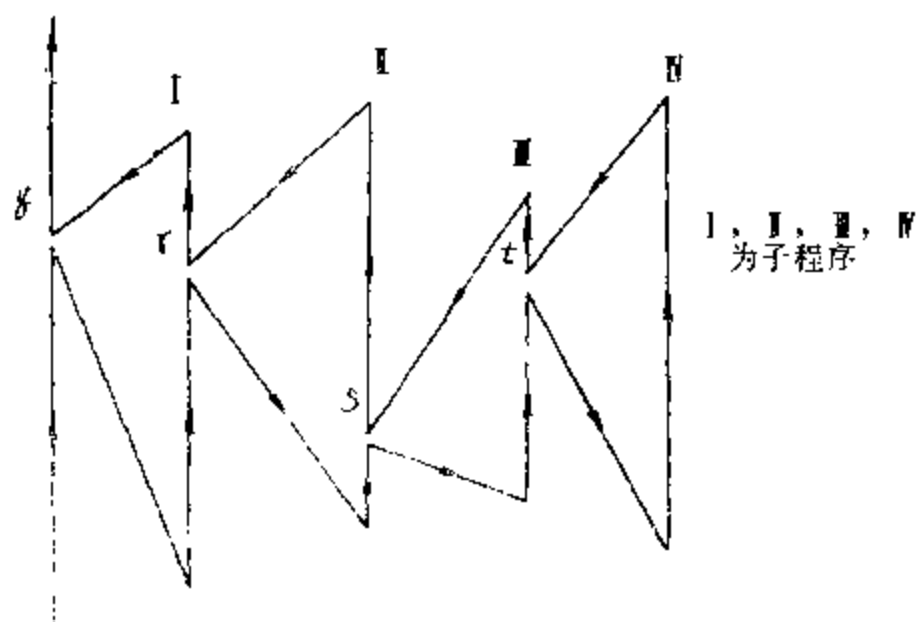


图 2.36 子程序的嵌套调用

因为子程序要求能嵌套调用和递归调用,所以只是有一般的、能把返回地址存在某个寄存器的转子指令(如前述 IBM370 的“转移与链接”指令)是不够的。例如,对于如图 2.36 的嵌套调用,若系统结构只提供保存返回地址于某个寄存器的支持,那在每个子程序之内都需有把这个寄存器内的返回地址存入指定给这个子程序存返回地址的某个主存单元的指令(一条或几条)。但这对递归调用(即子程序直接或经过别的子程序调用它本身,这在编译程序等系统软件中经常使用)还是不够的,因为对同一个子程序,在递归调用时需同时保存多个返回地址(其个数不能予知,取决于递归深度)。这样,就得在每个子程序内保留个数不定的用于存返回地址的工作单元。但是,若是把所有返回地址(q、r、s、t等)存在统一的返回地址堆栈,如图 2.37 所示,每次 CALL 时,把当时调用者的返回地址压入,在 RETURN 时执行弹出,那在每次 RETURN 时,在栈顶处的返回地址就是应有的返回地址。显然,采用返回地址堆栈这种办法会使嵌套调用和递归调用的返回地址处理简单得多,这样、就希望“转子”机器指令下是把返回地址送入某个寄存器,而是将它压入堆栈。大家知道, PDP-11 及其它一些机器和微处理器就有这种指令。



图 2.37 返回地址堆栈

然而,程序调用所需执行的操作远不只是转移到子程序首地址,并把返回地址存入堆栈如此简单。首先,需保存下来的调用者的信息不只是返回地址,还应有如条件码这样的状态信息以及某些寄存器的内容。其次,调用者需把参数传送给子程序。还有,需给子程序分配一个工作区,用以存局部变量和中间结果等。并且,由于前述嵌套调用和递归调用的要求,在每次执行 CALL 时,应把上述返回地址、状态位、寄存器内容、传送给子程序的参数、子程序的局部变量和中间结果等数据记录都压入堆栈,这样就能防止各个进程的这些数据在递归调用时被破坏。当然,在子程序出口(返回)时,不只是需把有关信息,如运算结果,送回调用者并把返回地址弹出堆栈,而且还需把上述数据记录都从堆栈中删去。

为程序调用的上述操作提供更好的支持是堆栈机器系统结构的一个显著特点，也是它与只有某些堆栈功能的机器（如 PDP-11）的明显区别。下面结合 HP-3000，具体地讲述堆栈机器的这个特点。

4.3-2 程序调用指令和返回指令

为了程序的递归调用，子程序应具有可再入性；这样，它在被某个进程调用，而还未执行完时，又可被另一个进程所调用。这当然要求子程序（或程序模块）本身在执行过程中不准改变。HP-3000 为了实现这点，使程序的指令代码与数据分开存贮，整个程序的指令代码按功能分段，各段是逻辑实体，称为代码段（关于程序的分段概念详见第五章）。

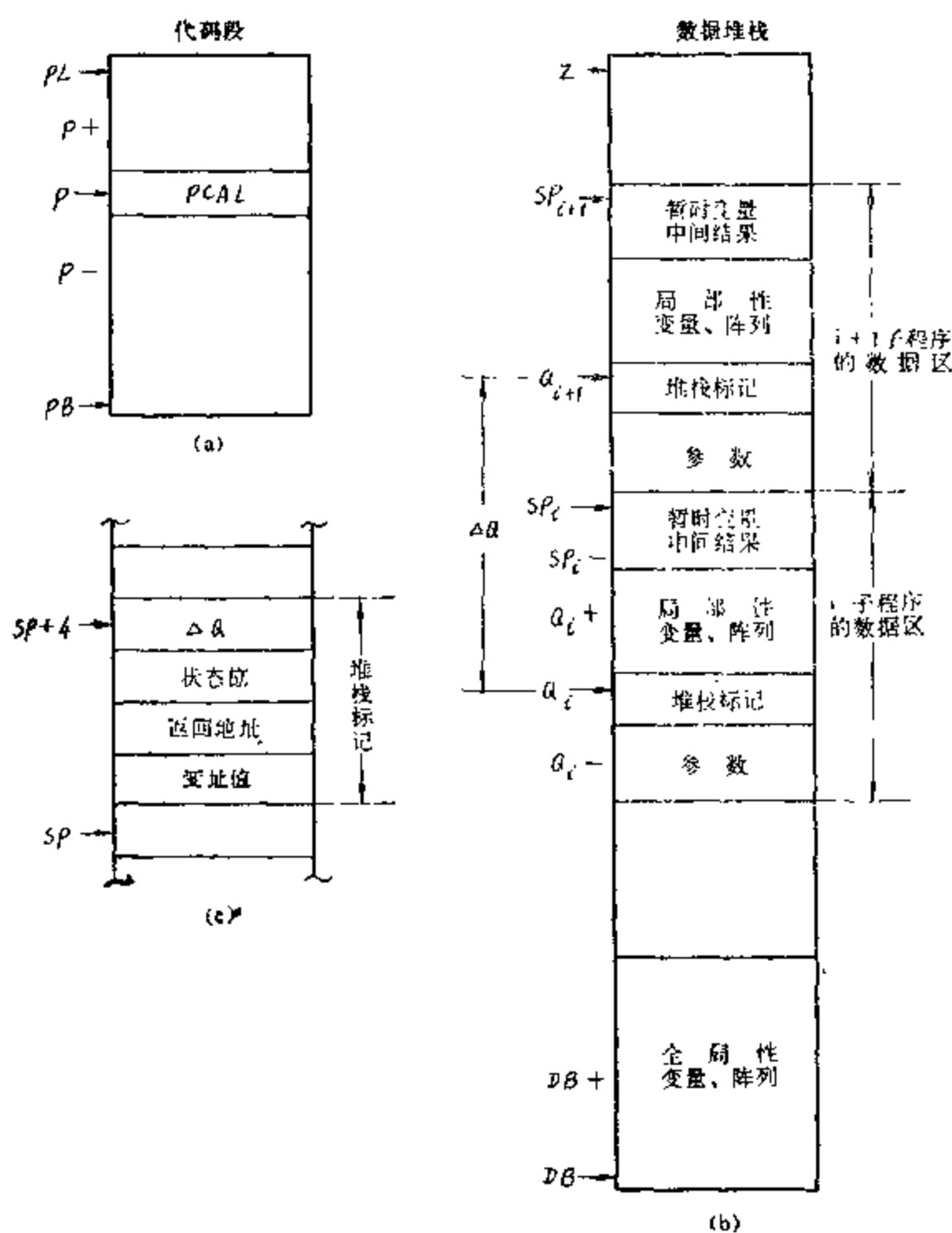


图 2.38 HP-3000 的代码段和数据堆栈

- (a) 代码段
- (b) 数据堆栈
- (c) 堆栈标记

正在执行的代码段（可看成是指令堆栈）由 PB 、 PL 和 P 寄存器指明，如图 2.38 (a)

所示。P 指向现行指令的地址，PL、PB 是这个段在主存中的上、下限地址，它们之间的距离随各段大小的不同而异。机器硬件保证在 PB、PL 之间的代码，只能被读出，不能写入，这是程序可再入性的要求。段内的转移指令的转向去址都是相对编址（相对于现行 P 值），因此代码段可重定位于主存的任何位置。

每个进程有一个在主存内 DB 和 Z 之间的数据堆栈，从子程序角度看，它的构成如图 2.38(b) 所示。全局性变量和阵列数据是所有子程序都可能要用到的，在整个程序的执行期间都需保持不动，而且要使所有子程序都能很快访问到，变量是相对 DB 直接编址，而阵列数据是相对 DB 间接编址。

调用每个子程序（如 i 子程序）时，它在数据堆栈内有一个区，它由“参数”、“堆栈标记”、“局部性变量和阵列数据”及“暂时变量与中间结果”组成。“参数”是需由调用者提供的参数；“局部性变量和阵列数据”是只为这个子程序所用的变量和阵列数据；SP_i（就是图 2.33 的 SP）指向“暂时变量与中间结果”的顶端；“堆栈标记”的构成如图 2.38(c) 所示，其作用下面再讲。

如前面 § 4.2-2 所述，SP_i 指向栈顶，其位置随运算时的压入和弹出而变化，运算数据可取自栈顶和次栈顶。然而，这种访问只适合于对中间结果的访问，访问不到在数据堆栈内的上述那些参数、变量和阵列数据等。因此，HP-3000 设置了前面讲过的“访存指令”，它有一个操作数是经地址访问。在数据堆栈内的数据都是相对于 DB、Q、SP 编址，所以需经相对地址访问。HP-3000 的地址码为十位，对于相对地址，它的组成如图 2.39 所示。用 § 3 讲过的扩展编码法指明是相对于哪个寄存器编址，剩下的位数 D 是相对量。P+、P- 是用于上述转移指令的转向去址。

方 式	b ₉	b ₈	b ₇	地 址 码	b ₂	b ₁	b ₀	访 存 有 效 地 址		
P +	0	0	←	-----	D	-----	→	P + D		
P -	0	1	←	-----	D	-----	→	P - D		
DB +	1	0	←	-----	D	-----	→	DB + D		
Q +	1	1	0	←	-----	D	-----	→	Q + D	
Q -	1	1	1	0	←	-----	D	-----	→	Q - D
SP -	1	1	1	1	←	-----	D	-----	→	SP - D

图 2.39 HP-3000 的访存相对地址

看出，对数据堆栈内的数据，既能经栈顶访问，又能用地址访问，这就大大增加了使用的灵活性，使得子程序在执行时能立即用硬件访问到属此子程序本身的参数、变量和阵列数据以及全局性变量和数据。然而，在子程序嵌套调用（这是经常要用到的）时，若有些数据是存在现行子程序区之前的调用子程序区内，则 HP-3000 是不能用硬件立即访问到，而需经软件往回查，才能取到。为这后一种访问或查找提供什么样的支持，是不同堆栈机器的差别标志之一。

现在讲述在 HP-3000 机器上，程序调用的具体执行过程。设正在执行的是 i 子程序[见图 2.38(b)]，此时数据栈顶在 SP_i，需调用 i+1 子程序。为此，先用 LOAD 指令把被调

用子程序所需的参数压入数据堆栈；压完后，SP 指向 $i+1$ 子程序的“参数”顶，接着执行 PCAL (Procedure Call 程序调用) 指令。

程序调用指令是一条功能很强的复合指令，它先把反映调用者 (i 子程序) 环境的“堆栈标记”[见图 2.38(c)] 压入堆栈。堆栈标记中，“变址值”是调用者的现行变址寄存器的内容；“返回地址”是相对 PB 编址 [参看图 2.38(a)]，这样在返回时，就是代码段在主存的位置变了也仍然能正确返回；“状态位”反映当时 CPU 的状态，如条件码等。PCAL 指令还改变 Q 寄存器的值，使之等于 ΔQ 所在单元的主存地址；对每个子程序，Q 恒指向堆栈标记的最高单元 (存 ΔQ 处)， ΔQ 是调用者与被调用者的 Q 值差。由 ΔQ 很容易用软件求得调用子程序的 Q 值，从而就能经 $Q+$ 、 $Q-$ 相对编址访问到调用子程序的变量和参数；当然，对于多级嵌套，可用各个子程序的 ΔQ 一直往回查。

PCAL 指令的地址码 (相对 PL 编址) 不是直接指向 $i+1$ 子程序的首地址，而是指向段表对应此子程序的入口。若由段表查得 $i+1$ 子程序是在现行段内，那就只需改变 P 值，使之指向此子程序的入口就行；若查得不是在现行段内，就需先改变 PB、PL 值使之对应到有该子程序的段，再变 P 值；若该段还在辅存，没调入主存，那就得先把它调进主存。上述操作都是由 PCAL 的微程序实现。

执行完 PCAL 指令，还得用 LOAD 指令把被调用子程序的局部性变量和数据压入堆栈；而后，才开始执行子程序。

在整个 $i+1$ 子程序执行完后，由返回 (出口，EXIT) 指令按 $i+1$ 子程序堆栈标记中的“返回地址”控制返回到 i 子程序 (即 P 需指向此“返回地址”)；EXIT 需执行 PCAL 指令上述那些操作的逆过程 (如使 PB、PL 指向有 i 子程序的段等等)。还需把数据堆栈内对应 $i+1$ 子程序的数据区弹出 (删去)； Q_{i+1} 之上的当然全部要删去，而 Q_{i+1} 之下要删去多少才能回到 SP_i ，则是由 EXIT 指令指明。

显然，为数据堆栈的上述构成和使用提供的这些支持比之于只是构成图 2.37 那样的返回地址堆栈，大大简化了对子程序嵌套调用的管理；例如，给各个子程序分配工作区的管理就要简单得多，在子程序用完后，它的工作区能自动腾出。

可见，堆栈机器的系统结构设计是深入于程序的具体执行过程，这是它的显著特点。也正因为如此，对于同样的操作功能，它的操作系统可以比非堆栈机器的小。

4.4 堆栈型机器与通用寄存器型机器的简单比较

由前面的讲述可以看出，堆栈机器与一般通用寄存器机器的差别远不只是堆栈机器是所谓的零地址机器，从而能省去地址码，缩短指令的长度。它们之间的根本区别在于堆栈机器为软件中广泛应用的堆栈数据结构提供直接的有力支持，这有助于缩小高级语言与机器语言在语义上的差别，也使得操作系统中用得很多的堆栈数据结构更易于实现，使得高级语言和操作系统都能更高效地在机器上实现。

前面讲了，在堆栈机器上能直接由逆波兰表示式形成机器语言程序。这样就能以逆波兰表示式作为编译时的中间语言，当然也就不必先把源程序编译成汇编语言程序，再翻译成机器语言程序。更由于堆栈机器对程序递归调用的支持，就使得多种语言 (如 ALGOL, PASCAL, PL/I 等) 的编译程序的组成比通用寄存器型机器的简化，而且支持了编译程序本身用被编译的语言编写。用 ALGOL 语言编写 ALGOL 编译程序就是首先在堆栈机器

B-5500 上实现的。

至于堆栈机器在支持子程序嵌套、递归调用方面的优点，那是通用寄存器型机器肯定比不上的。堆栈机器能方便、快速地建立起被调用子程序的局部环境；如 HP-3000 那样，子程序的局部变量和参数都是相对于 Q 寄存器被访问。Q 寄存器可以称之为环境指针寄存器，它和通用寄存器型机器的基址寄存器虽有相似之处，然而基址值是需由单独的机器指令置定，可是 Q 值却是在执行 PCAL 指令时，按当时的 SP 值由硬件改变，并能把与原有 Q 值之差 ΔQ 自动存在堆栈。这当然简化了程序的调用，而且不必如一般通用寄存器型机器（如 IBM370）那样，在指令中需设置“基址寄存器”字段。

其实，在通用寄存器型机器上，ALGOL 和 PASCAL 等语言的实现以及子程序的调用等都应用了堆栈数据结构，只不过系统结构没有相应的堆栈数据表示而已。

还有，由于堆栈机器的访存地址几乎都是相对型的（见图 2.39），因此访存地址码的宽度可比通用寄存器型机器的短。另外，由于通用寄存器的“通用”性，它的优化管理和使用就比较麻烦，这要增加编译程序的负担，从而要多用存贮单元，增长编译时间；而堆栈机器的寄存器则几乎都是专用的，且多是由硬件管理，减轻了软件的负担。并且，由于堆栈机器对所有中间结果和中间变量（如编译过程中只用到一次的所谓无名变量）都不必赋以地址，而是全压入堆栈。这样，既省了赋地址所需的指令，又由于在用完能自动被弹出，自动腾出已不用的单元，从而提高了存贮单元的利用率。

上述这些，使得程序的总位数和程序执行所需用的存贮单元都比通用寄存器型机器的少；有人认为，可减少 1/3 以上。亦即堆栈机器的存贮效率要比通用寄存器型机器的高。

此外，由于堆栈机器的相对型编址方式，为扩大地址空间并不需要增长指令的地址码长度，而只需增长诸如 P, DB, Q, SP 等寄存器的位数。

不过，对于堆栈机器能经逆波兰表示式直接形成机器语言的好处不宜强调过份。因为远不是每个表达式都具有如图 2.31 例子那么多的运算符。例如对 21,078 个表达式的统计，总共只有 15,969 个运算符，即每个表达式平均只有 0.76 个运算符；又如，对 440 个 FORTRAN 程序的统计也表明赋值语句中，有 60% 是除了“=”赋值符外，没其它运算符；对事务处理更是如此，对 120 个 PL/I 程序的统计表明 98% 的表达式至多只有一个运算符。显然，对于运算符个数少的表达式的实现，通用寄存器型机器和堆栈机器是一样的。从 HP-3000 上，编译程序、BASIC 解释程序、文本编辑程序等软件的运行统计来看，也说明这点。统计表明真正零地址的堆栈型指令只占 16%，而访存型指令却占 34%，其中单纯压入、弹出的 LOAD、STOR 指令为 25%。

对某些方面的应用，堆栈机器不如通用寄存器型机器。例如，虽然 B-6700 可以用栈顶的 32 个寄存器进行某些向量运算，HP-3000 也设置了一个变址寄存器，但堆栈结构本身却是向量、矩阵运算所用不上的，对于这些运算是通用寄存器型机器比堆栈机器更合适。

在运算速度上，目前的堆栈机器一般地是比通用寄存器型机器低。其主要原因在于前者的访存效率比后者的低，即堆栈机器在解题时的访存次数要多。例如，在具体运算前，往往需先将一批参数由主存压入堆栈，这需要访存；如果这批参数的数量超过栈顶寄存器的个数，那就得把访存已取来的参数压入在主存内的堆栈，这又得访存；当需对存在主存堆栈内的这些参数进行运算时，又得访存将它们取到栈顶寄存器。至于通用寄存器型机器，某条指令所要用到的操作数可直接取自主存任意单元和通用寄存器中的任一个；因此它的访存次数

可以较少。又如，对于需多处、多次使用的中间运算结果，通用寄存器型机器可以将它存在某个通用寄存器，从而不需经访存就能取得；而对堆栈机器，这种中间运算结果是很难于常存在栈顶寄存器，因而往往需经访存才能取得。应该说，访存效率较低的上述缺点并不是堆栈机器所必然具有的；它是最初设计堆栈机器时，从尽量简化硬件结构所引起的。显然，如果增加堆栈的个数（每个都有其各自的栈顶寄存器），使之对应于不同的用途，如设置存那些需多次使用的中间运算结果的堆栈等，并使一条指令的操作数可取自不同的堆栈，再加上增多栈顶寄存器的个数，那访存效率是会显著改善的。

从过去的历史看，通用寄存器型机器和堆栈型机器都得到了发展。它们之间不应是相互否定，而应是相互结合，近年来某些微处理器和小型机的发展就是这样，今后的机器看来更会是如此。

主要参考文献

- [1] G. J. Myers, "Advances in Computer Architecture," John Wiley & Sons, 1978.
- [2] K. J. Thurber and P. C. Patton, "Data Structures and Computer Architecture — Design Issues at The Hardware/Software Interface," D. C. Heath and Company, 1977.
- [3] D. J. Kuck, "The Structure of Computers and Computations," Vol. 1, John Wiley & Sons, 1978.
- [4] V. C. Hamacher, et al, "Computer Organization," McGraw-Hill, 1978.
- [5] J. P. Hayes, "Computer Architecture and Organization," McGraw-Hill, 1978.
- [6] A. S. Tanenbaum, "Structured Computer Organization" Prentice-Hall, 1976.
- [7] Yaohan Chu, (ed.) "High-Level Language Computer Architecture," Academic Press, 1975.
- [8] R. L. Sites, "Operating Systems and Computer Architecture," Ch. 12 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [9] W. M. McKeeman, "Stack Computers," Ch. 7 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [10] E. I. Organick and J. A. Hinds, "Interpreting Machines, Architecture and Programming of the B1700/B1800 Series," North Holland, 1978.
- [11] E. I. Organick, "New Directions in Computer Systems Architecture," Euromicro Journal, Vol. 5, No 4, July 1979, pp. 190—202.
- [12] D. R. Ditzel and D. A. Patterson, "Retrospective on High-Level Language Computer Architecture," The 7th Annual Symposium on Computer Architecture, 1980, pp. 97—104.
- [13] IBM 4300 Processors Principles of Operation for ECPS, VSE Mode, GA22-7070-0, 1979.
- [14] W. S. Brown and P. L. Richman, "The Choice of Base," Comm. of the ACM, Vol. 12, No 10, Oct. 1969, pp. 560—561.
- [15] E. A. Feustal, "On the Advantages of Tagged Architecture," IEEE Trans. on Computers, Vol. C 22, No 7, July 1973, pp. 644—656.
- [16] R. P. Case and A. Padegs, "Architecture of the IBM System, 370," Comm. of

- the ACM, Vol. 21, No. 1, Jan 1978, pp. 73—96.
- [17] D. M. Bulman, "Stack Computers, An Introduction," Computer, Vol. 10, No. 5, May 1977, pp. 18—28.
- [18] R. P. Blake, "Exploring a stack Architecture" Computer, Vol. 10, No. 5, May 1977, pp.30—39.
- [19] W. D. Strecker, "VAX—11/780—A Virtual Address Extension to the DEC PDP—11 Family," AFIPS Conf. Proc., Vol. 47, 1978, pp. 967—980.
- [20] W. T. Wilner, "Design of the Burroughs B1700," AFIPS Conf. Proc., Vol. 41, 1972, pp. 489—497.

第三章 控制方式

大家知道，有些高级语言的一条语句或是一个操作系统命令可以用一串机器指令或一段汇编语言程序来解释，而每条机器指令也是可以用一串微指令来解释的。因此，就可把解释看成是对作用于特定数据表示式（语句）的赋值，这里讲的表示式可以是高级语言的，也可以是低级语言的。

我们曾经多次讲过，虽然某些高级语言语句（如 BASIC 语句）目前是用一串机器指令去解释，但从原理上是完全可以直接由一串微指令来解释，又如中断的处理，既可以用一串机器指令实现，也可以直接由一串微指令实现。因此，我们在本章中虽然讲的是如何控制机器语言的解释过程，但其基本概念也适用于高级语言和中间语言的解释。机器语言和高级语言其实并没有截然的界限，而且，逐步减少机器语言和高级语言（包括操作系统控制语言）在语义上的差别是发展趋势。

机器语言的解释过程是由机器内的控制机构控制，它调用一串微指令（或微操作）去实现每条机器指令；此外，还控制这一串微指令与解释下条机器指令的另一串微指令间的衔接。当解释每条机器指令的微动作及其动作顺序定了后，机器语言程序的加快执行，可以通过控制机构同时解释二条、多条以至整段程序来达到，这里，重迭和流水是所用的基本技术。本章主要讲述这二种技术的具体实现及其性能分析。这是改进从机器设计者看的系统结构的一个主要方面，然而它往往也是为改进从程序设计者看的系统结构所必需的。基于上一章所讲的向量数据表示的向量机就是一例，没有流水技术的支持，光设置向量数据表示是没有实际意义的。为了发挥向量机所能提供的效能，还必须改进所用软件，这是系统结构的改进会对软件设计提出新要求的一个例子。关于向量机我们在 § 2 要讲到。

最后，简述一下分布处理的控制方式。

§ 1 重迭方式

从速度来看，指令的解释可以有三种方式，即顺序方式、重迭方式和流水方式。顺序方式在计算机原理中已经学过，是大家熟悉的，这里只简单的提一提。

1.1 顺序解释方式

顺序方式可以有两层意思，一是各条机器指令间是按顺序串行执行，二是一条机器指令内的各个微操作是按顺序串行执行。指令间的顺序执行是在执行完一条指令后，才取出下条指令来执行。至于指令内各个微操作的串行执行，以大家熟悉的加法指令为例，指的是取现行（加法）指令，指令译码，形成操作数真地址并取操作数，执行相加并存贮结果，形成下条指令地址等都是顺序串行执行。

顺序执行的优点是控制简单，其缺点当然是速度上不去，因为在上一步操作完成前，下一步不能开始。另外，机器各部件的利用率也会因此而不高。例如，在取指令和取操作数期间，主存贮器是忙碌的，但是运算器却在空闲着。在对操作数执行运算期间，运算器是忙碌的，主存却在空等。

对于顺序方式，由于下条指令的地址是在指令解释过程的末尾进行，因此不论是由指令

计数器加 1，还是由转移指令把转向地址送到指令地址计数器，去形成下条指令地址，由本条指令转入下条指令的时间关系都是一样的。

1.2 重迭解释方式

早在五十年代初期，UNIVAC-1 计算机就已有了中央处理机、存贮器和输入输出设备之间的重迭操作，即 CPU 启动 I/O 后，CPU 和 I/O 可以同时操作，虽然这也可看成是一种重迭方式，然而，它只能实现 I/O 指令与其它指令的重迭。我们把这种重迭放在下一章“输入输出系统”中讲述，而在这里讲述的是如何实现非 I/O 指令的微操作的重迭。

可以把前面讲过的，解释一条机器指令的那些微操作归并成取指令、分析与执行，如图 3.1 所示。指令的分析包括对指令的译码、形成操作数真地址并取操作数以及形成下条指令地址等；指令的执行则指的是对操作数的运算并存贮运算结果。

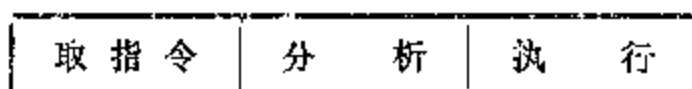
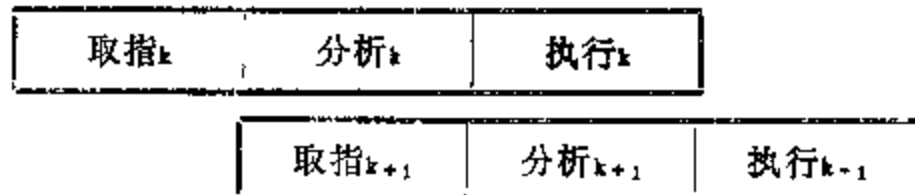


图 3.1 对一条机器指令的解释

这样，指令的顺序解释就是在第 k 条指令的执行操作完成之前，解释第 $k+1$ 条指令的任何操作都不开始执行，如图 3.2(a) 所示；而指令的重迭解释则指的是在解释第 k 条指令的操作完成之前，就可开始解释第 $k+1$ 条指令，图 3.2(b) 是可能的一种方式。显然，重迭解释并不能加快一条指令的实现，但能加快相邻二条指令以至一段程序的解释。



(a) 顺序解释



(b) 重迭解释的一种方式

图 3.2 指令的顺序解释与重迭解释

下面来看看为实现这种“分析_k”和“取指_{k+1}”的重迭对硬件结构应提出什么要求。

“分析_k”的主要操作是形成操作数真地址和访存取操作数，而“取指_{k+1}”当然也需访存；但是一般的机器，操作数和指令是混合存贮于同一主存内的，而且主存同时只能访问一个存贮单元，这样，就实现不了“分析_k”与“取指_{k+1}”的重迭。因此，需要采取相应的措施。

一种办法是使操作数和指令分别存于二个独立编址且可同时访问的存贮器。已有的一些专用机和通用机（如 HP-3000 等）就是这样安排的。这种方法需增加设备，使总线的控制复杂些，而且可能给软件设计带来某些不便。

另一种办法是指令和操作数仍然混存，但采用多体交叉主存结构（见第五章 §1.3），若第 k 条指令的操作数与第 $k+1$ 条指令不是存于同一个体内，则能在一个主存周期（或少许多一些时间）内取得这二者，从而可实现“分析_k”与“取指_{k+1}”重迭；当然，若这二者正好共存于一个体内，那就做不到重迭。

还有一种办法是设置指令缓冲寄存器（简称指缓），予先把指令由主存取到这个寄存

器。这样，“分析_k”就能与“取指_{k+1}”重迭，因为只是前者需访主存取操作数，而后者是由指令缓冲寄存器取第 $k+1$ 条指令。由于一条指令的“执行”操作时间往往会比“分析”的长，那在“分析”与“执行”重迭时，主存是会有空闲的时候，利用它就可把指令予取到指令缓冲器。这样一来，若每次在“取指”时都可由“指缓”取得已从主存“予取”得的指令，则“取指_{k+1}”只需很短时间就可完成，所以可合并到“分析_{k+1}”内，从而构成如图 3.3 所示的一次重迭方式。

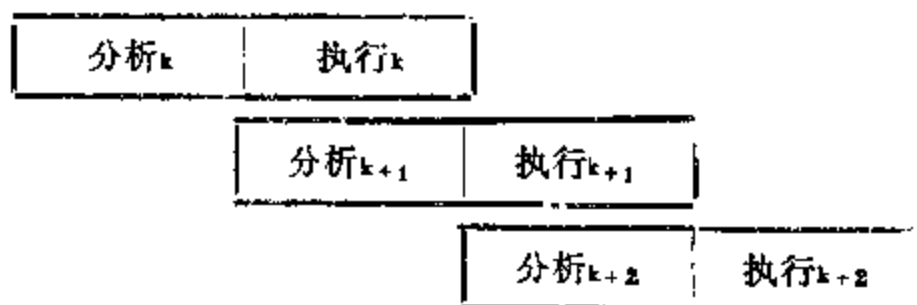


图 3.3 一次重迭工作方式

所谓“一次重迭”是指的任何时候都只是“执行_k”与“分析_{k+1}”重迭。就是说，当“分析_{k+1}”比“执行_k”提前结束时，“执行_{k+1}”也不接着“分析_{k+1}”在“执行_k”结束前进行；同样，在“执行_k”比“分析_{k+1}”提前结束时，“分析_{k+2}”也不接着“执行_k”在“分析_{k+1}”结束前进行。因此，这种重迭方式的机器内只需有一套指令分析部件（包括形成操作数地址的加法器）和指令执行部件（如运算器等），这也有助于简化控制。一次重迭是重迭机器一般采用的办法，我国 DJS-240 机也是采用这种方式。

当然，为要实现“分析”与“执行”重迭，还需解决好控制上的好多问题。

例如，若第 k 条指令是按其执行结果进行转移的条件转移指令，且成功转移到 l 单元，则与“执行_k”重迭的“分析_{k+1}”当然是白做了，需要撤消并从头分析第 l 条指令。一种可能的处理方式如图 3.4 所示。当然，若第 l 条指令还没取到指缓内（一般重迭机器就是如此），则在“执行_k”之后，还需先进行“取指 l ”才能“分析 l ”。

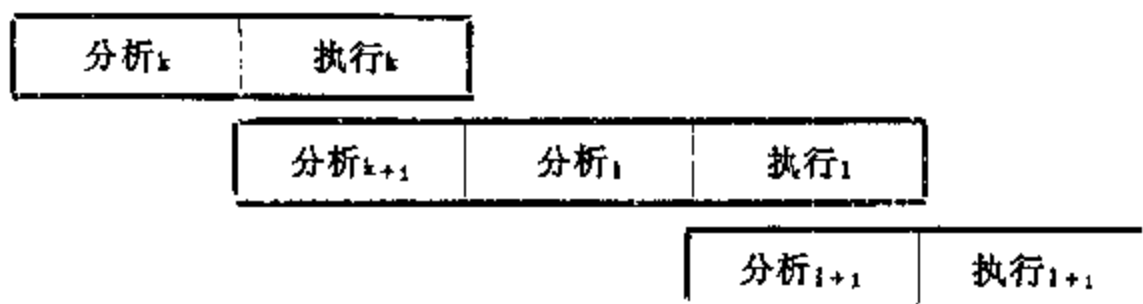


图 3.4 转移成功时的重迭处理

又如，若第 $k+1$ 条指令的操作数地址 i 正好就是第 k 条指令存放运算结果的地址，这在顺序解释时，由于是先由第 k 条指令把运算结果存进主存 i 单元，而后再由第 $k+1$ 条指令从 i 单元取出，当然不会出错；但在上述重迭解释时，由于“分析_{k+1}”与“执行_k”重迭，在“分析_{k+1}”时从 i 单元取出的内容若是“执行_k”存进运算结果前的原存内容，而不是应有的第 k 条指令的运算结果，则必然要出错。我们把出现上述情况称为出现“数相关”。

所谓“相关”，指的是由于一段机器语言程序的相近指令之间出现了某种关联，使得它们不能同时被解释。上面例子是在第 k 、 $k+1$ 指令的数地址之间有了关联，即出现数相关，因而不能同时解释它们。当然，数相关不止是会发生在主存空间，也会出现于通用寄存器空间，这点在下面还要详细分析。

除“数相关”外，还有“指令相关”。例如，若出现如下情况：

	源地址	目的地址
$k-1$:	存	3
k :		k

“3”为通用寄存器号，对一次重迭，这种情况就出现指令相关，第 $k-1$ 、 k 这二条指令不能同时解释，因为只有在第 $k-1$ 条指令执行完之后，取出的第 k 条指令才是应有的。

特别是在设置有指令缓冲器时，由于指令是插空予取进指缓，若指缓可存 l 条指令，则指令相关在 $(k-l)$ 到 k 指令之间都可能发生，这是由于在执行第 $k-l$ 条指令时，从第 $k-l+1$ 直至 k 指令可能都已予取进指缓。显然，指缓容量愈大，或是说指令予处理能力愈强的机器出现指令相关的概率愈高。

下面以 DJS-240 机为例，分析重迭机器如何处理相关问题，以使大家对相关处理能有具体的认识，这对学习下一节要讲的流水处理是有好处的。

1.3 DJS-240 机的相关处理

DJS-240 机是中型通用机。它的中央处理机由分析器（其功能包括指令译码、操作数地址形成和操作控制等）、指令缓冲寄存器、通用寄存器组（ 16×32 位）、主存控制器（简称存控）和运算器（包括有运算用寄存器、加法器、移位器、乘法器和运算控制器）等组成。分析器、运算器由各自的微程序控制器控制，采用同步控制方式。除中央处理机外，还有通道、主存、外设和外设控制器等，其结构与第一章图 1.15 上所示的相同。

1.3-1 指令相关的处理

由上面的分析可知，对于有指令缓冲器的机器，由于指令是提前由主存取进指缓，因此，为了指令相关的处理，需要进行相当复杂的多条指令地址与多条指令的运算结果地址相比较；此外，在相关时还需控制更改指缓内容。

那么，能否从根本上消除“指令相关”的出现呢？显然，若在程序运行过程中不准修改指令，就不会出现指令相关，DJS-200 系列机就是如此安排。当然，不准修改指令的更主要原因还在于这很有利于实现程序的可再入性及程序的递归调用。

如果非要在程序运行过程中修改指令，DJS-200 设置了和 IBM370 一样的“执行”指令（详见第二章 § 3.2-1，参看图 2.21）。由于可能被修改的指令是以“执行”指令的操作数形式出现，所以指令的这种允许修改并不会导致出现指令相关，而只可能以数相关形式出现。这样，一次重迭的 240 机就不必考虑指令相关问题。关于数相关，下面先讲 240 机如何处理主存空间的数相关。

1.3-2 主存空间数相关的处理

240 机是通过给读数、写数申请安排不同的访存级别来解决主存空间的数相关问题。

240 机是中央处理机和通道（不止一个）都可直接访问主存，因此，需要由存控对各方面来的访存申请进行安排。访存申请可分成取指令、读数、写数、（前三者是由中央处理机发出的）和通道申请（包括读、写数据，取通道控制字、存通道状态字等等）四种。这四种申

请可能同时发出,致使出现访存冲突,这需由存控按预定的优先级别排队处理。DJS-200系列机规定的优先次序为:通道申请、写数、读数、取指令。

当第 k 、 $k+1$ 条指令出现主存数相关时,“执行 k ”与“分析 $k+1$ ”同时对同一个单元发出访存申请,如图 3.5(a)所示。但因“写数”的级别高于“读数”,则存控必是先执行写数,而后再执行读数。若“分析 $k+1$ ”设计成是读数申请不被响应时,自动延长到被响应为止,则“执行 k ”与“分析 $k+1$ ”重迭的实际时间关系如图 3.5(b)所示。看出,DJS-240 实际上是由存控来处理主存空间的数相关。

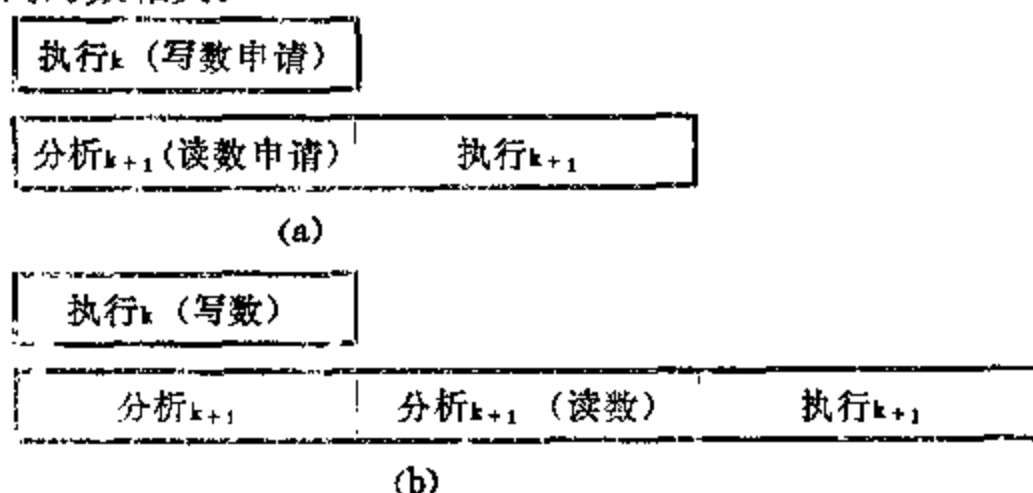
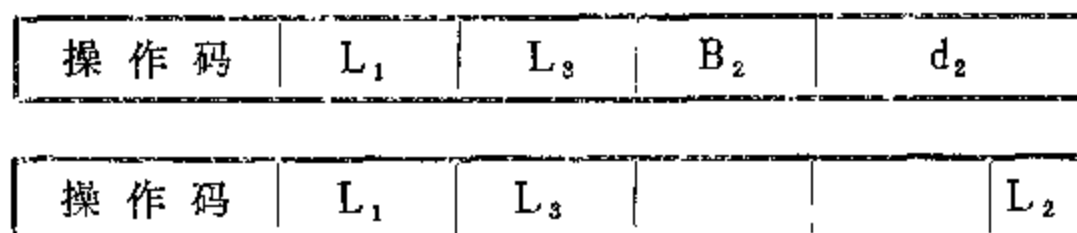


图 3.5 240 机对主存数相关的处理

1.3-3 通用寄存器组的相关处理

在第二章 § 2.2 已经讲过, DJS-200 系列机的基本指令格式为:



L_1 、 L_3 指明存第一操作数和运算结果的通用寄存器号; B_2 为形成第二操作数地址的变址值所在的通用寄存器号。就是说,通用寄存器既存操作数、运算结果,也存变址值。由于变址值是在“分析”时用,操作数是在“执行”时用,而运算结果一般是在“执行”的末尾才形成,因此变址值相关和数相关的状况和处理方法是不同的,下面分别分析之。

一、“数相关”

240 机设计成是在“分析”周期由分析器控制,从通用寄存器组(16 字×32 位)取操作数送入运算器的相应 B、C 寄存器,供下一个“执行”周期用。240 机的典型“分析”、“执行”周期在正常情况时为 4 拍,等于主存周期。由于有些指令需由通用寄存器组取两个操作数: (L_1) 和 (L_2) ,而通用寄存器组的结构是每次只能读出一个数,所以 (L_1) 和 (L_2) 需在不同拍取得送入 B、C。运算结果是在“执行”的末尾送入通用寄存器。这样,“执行 k ”与“分析 $k+1$ ”访问通用寄存器组的时间关系如图 3.6 所示。

看出,当出现 L_1 相关,即 $L_1(k+1) = L_3(k)$ 或 L_2 相关,即 $L_2(k+1) = L_3(k)$ 时,在“分析 $k+1$ ”取来的 (L_1) 或 (L_2) 就是错误的。

显然,解决 L_1 、 L_2 数相关的一种办法是和前述处理主存数相关一样的推后“分析 $k+1$ ”法。当然,若把“分析 $k+1$ ”完全推后到“执行 k ”执行完才开始,那在数相关时,一次重迭就变成顺序串行执行,影响了速度。由图 3.6 可看出,在相关时只要推后到是“执行 k ”先

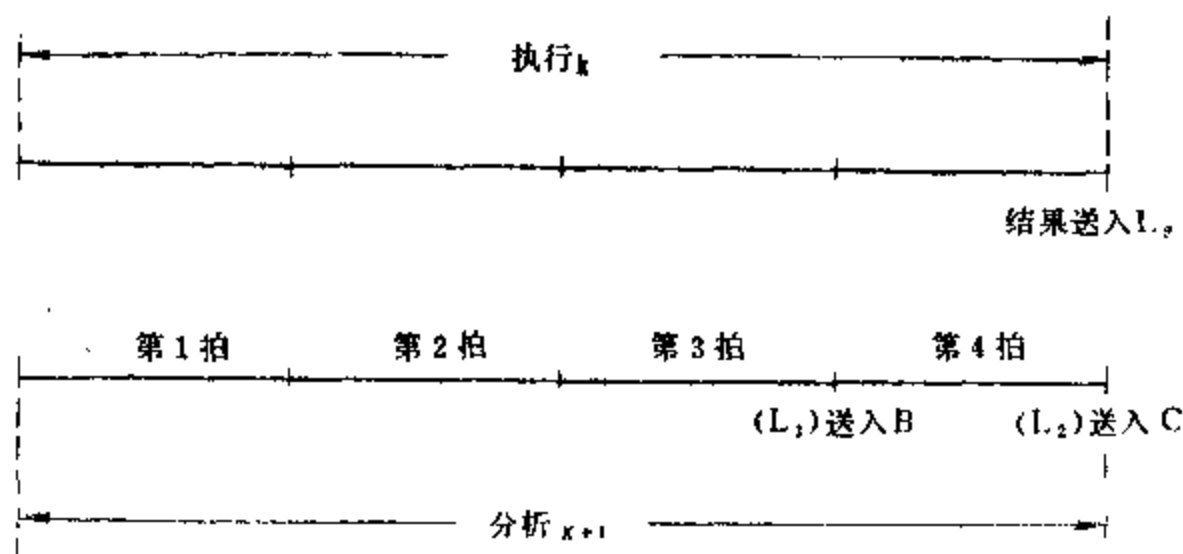


图 3.6 “执行 k ”、“分析 $k+1$ ”重迭时，访问通用寄存器组的时间关系

把结果送入 L_3 ，再由“分析 $k+1$ ”去取 (L_1) 或 (L_2) 就行。这样，就仍然可实现“执行 k ”和“分析 $k+1$ ”的部分重迭以减少速度损失。至于在 L_1 、 L_2 相关时各需推后多少拍，这和所用通用寄存器的地址译码机构和读、写机构有关，是具体实现方法的问题，这里就不讲了。

应该说，出现 L_1 相关的概率还是较高的。例如，若某个通用寄存器是作为累加器使用，则本条指令存进去的运算结果往往在下条指令又得作为操作数取出来使用。那么，是否有办法在出现 L 相关时，能不用推后“分析 $k+1$ ”法呢？增设“相关专用通路”是常用的解决办法。

所谓“相关专用通路”法是指的在出现 L_1 、 L_2 相关时，不是按

结果 $\rightarrow L_3 \rightarrow B(或C)$ 寄存
(写入) (读出)

器而是经专用通路直接把运算结果送回 B 、 C 寄存器，即

结果 $\rightarrow B(或C)$ 寄存器
(直接)

其具体实现如图 3.7 所示。这样，不论 L_1 或 L_2 相关，只需在“执行 k ”的末尾，既把运算结果送入通用寄存器，还同时把相关操作数送入 B 或 C 寄存器，从而在相关时不必推后“分析 $k+1$ ”，改变“执行 k ”与“分析 $k+1$ ”的相互时间关系。

“相关专用通路”方法是维持原有重迭时间关系，用增加设备来解决相关问题，而前述推后“分析 k ”则是基本上不增加设备，以降低速度为代价来解决。这二种方法是解决重迭机器相关问题的基本办法。显然，若某种相关的出现概率低，那就不应该用“相关专用通路”法。

虽然“相关专用通路”方法也可应用于解决前面讲过的主存空间的数相关，在概念上只需把图 3.7 的通用寄存器组改为主存。但由于 240 机是一次重迭，主存数相关的出现概率要

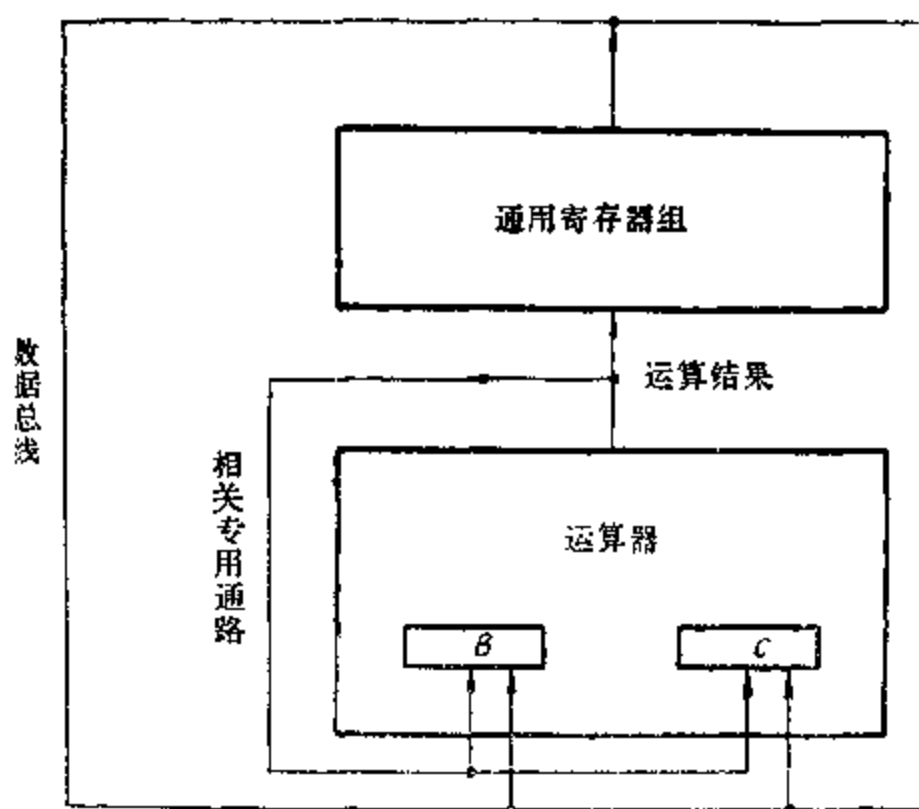


图 3.7 用相关专用通路解决通用寄存器组的数相关

比通用寄存器组数相关的出现概率低得多, 因此就不需要采用“相关专用通路”法。

二、“变址值相关”

我们在上一章 § 2.2 讲过, DJS-200 系列机的访主存第二操作数的有效地址是如下形成:

$$(X_d) + (B_2) \cdot (B_2 \neq 0) + d_2$$

这当然需经加法器执行, 240 机是由分析器内的地址加法器实现的。由于“分析”周期的正常值等于主存周期, 所以, 从时间关系上要求在“分析”周期的始点就能由通用寄存器输出总线取得 (B_2) 送入地址加法器。由于运算结果是在“执行”周期的结尾才送入通用寄存器组, 它当然不能立即出现在通用寄存器输出总线上, 也就是说, 由图 3.8 可看出, 在“执行_k”得到的送入通用寄存器的运算结果是来不及为“分析_{k+2}”作变址值用。这样, 虽然 240 机是一次重迭, 但对变址值相关 (B 相关) 则不止会出现一次相关, 还会出现二次相关。即当出现 $L_3(k) = B_{(k+1)}$ 时, 称为出现 B 一次相关; 而当出现 $L_3(k) = B_{(k+2)}$ 时, 称为出现 B 二次相关。

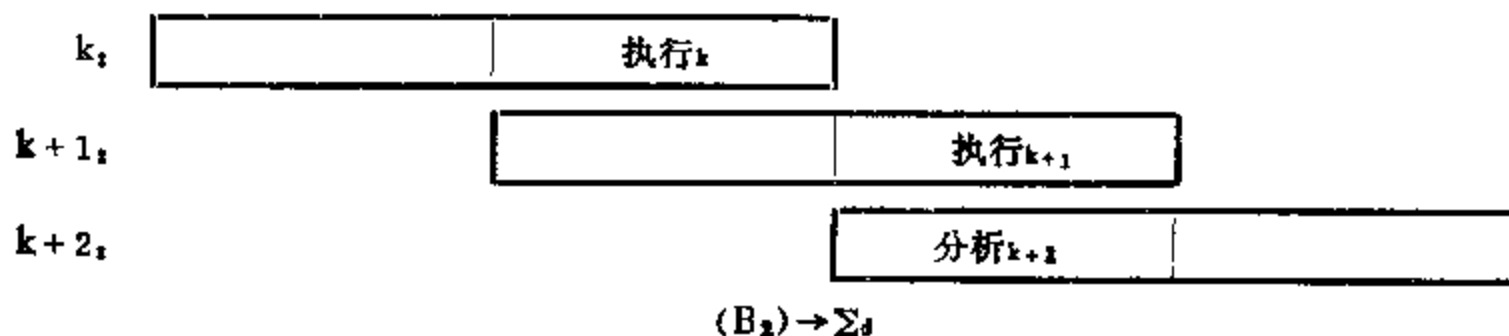


图 3.8 B 一次相关与二次相关

对于 B 相关, 其解决办法同数相关一样, 也可有推后“分析”和“相关专用通路”二种。

先讲推后“分析”的方法。由图 3.8 可看出, 对于 B 二次相关, 只需推后“分析_{k+2}”的始点, 使得“执行_k”时送入通用寄存器的运算结果, 在“分析_{k+2}”开始时已可出现于通用寄存器输出总线上, 如图 3.9(a) 所示。至于推后多少拍, 这取决于通用寄存器组译码、读出机构的具体逻辑组成。而对 B 一次相关, 则除此之外, 还需再推后一个“执行”周期, 如图

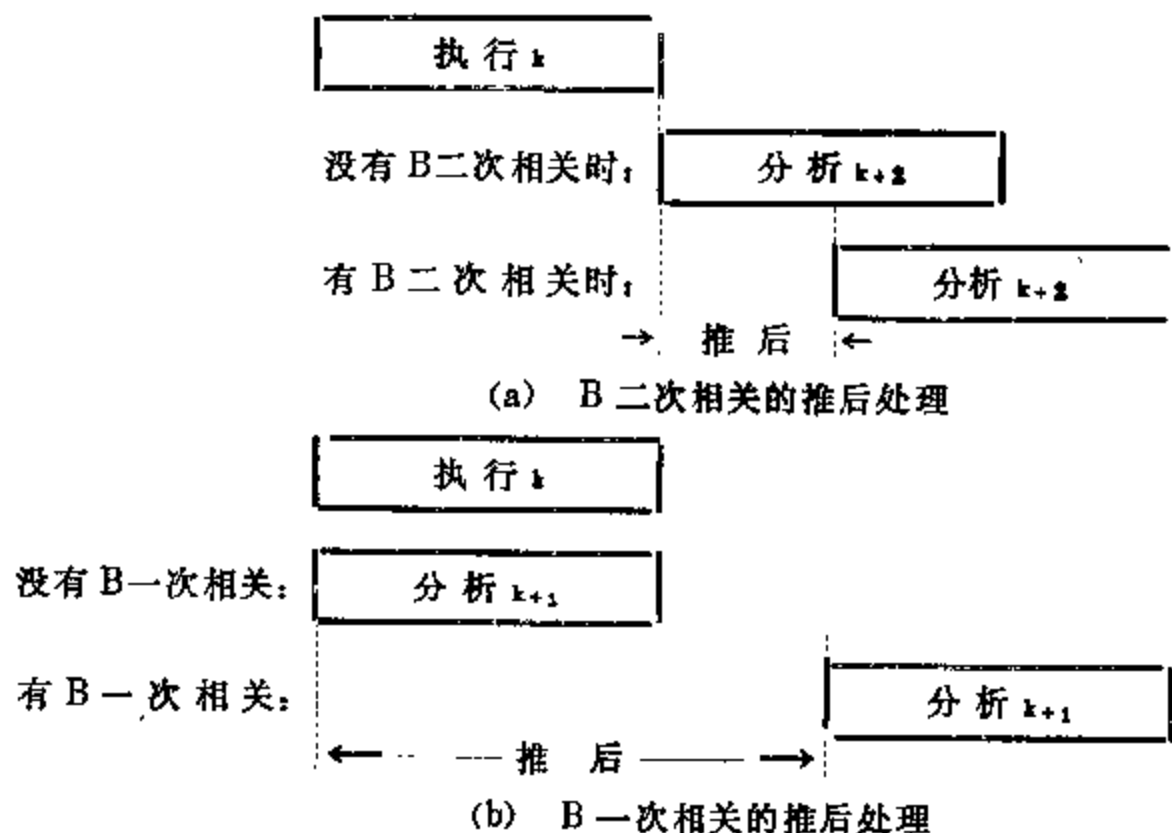


图 3.9 B 一次、二次相关的推后处理

3.9(b)所示。

由于 B 一次、二次相关的概率并不很低，增设 B 相关专用通路还是值得的，其办法与图 3.7 的相似，如图 3.10 所示。这样，

在 B 二次相关时，就可把“执行_k”得到的运算结果，在送入通用寄存器组的同时，经“B 相关专用通路”直接送到“访存操作数地址形成”机构，从而不必推后“分析_{k+2}”。同理，在 B 一次相关时，只需使“分析_{k+1}”推后到接着“执行_k”进行就可以了。

综上所述，为了实现同时解释二条指令，即实现在时间上的重迭，需要付出空间代价，如增加数据总线和控制总线；增设地址加法器等。此外，一定得处理好这二条指令之间可能存在的关联，如转移处理以及指令相关、主存操作数相关和通用寄存器空间相关的处理等等。还有，在设计时应使重迭的“分析”和“执行”周期尽可能等长；这样，分析器、运算器才能都不间断地运行。

显然，若要同时解释更多条指令，那转移问题和相关问题的处理必然要复杂得多。知道了 DJS-240 机一次重迭是如何具体处理的，那在下面讲“流水方式”时，一些具体实现上的问题就可以不讲了。

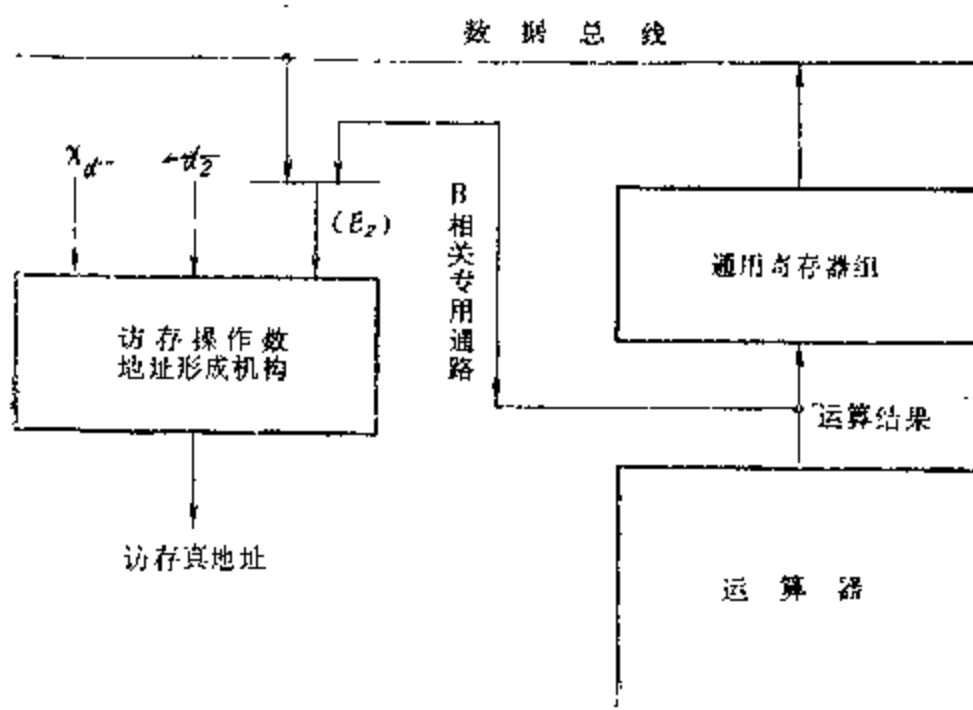


图 3.10 B 相关专用通路法

§ 2 流水方式

本节主要从控制角度来分析流水结构。流水方式在巨型机和大型机中已普遍采用，而且八十年代的中、小型机也很可能会采用。本节先讲述流水方式的基本概念，接着分析流水结构的性能，而后讨论流水方式的相关情况及其处理方法，最后简述采用向量数据表示和流水技术相结合的向量流水处理。

2.1 基本概念

2.1-1 从重迭到流水

上一节讲的“分析_{k+1}”与“执行_k”的重迭如图 3.11 所示，可以看成是把一条指令的解释过程分解成“分析”与“执行”二个子过程。由于这二个子过程是分别实现于分析器和运算器这二个独立部件，所以就不必等待上一条指令的“分析”、“执行”子过程都完成后才送入下一条指令，而是可以在上一条指令的“分析”子过程结束，

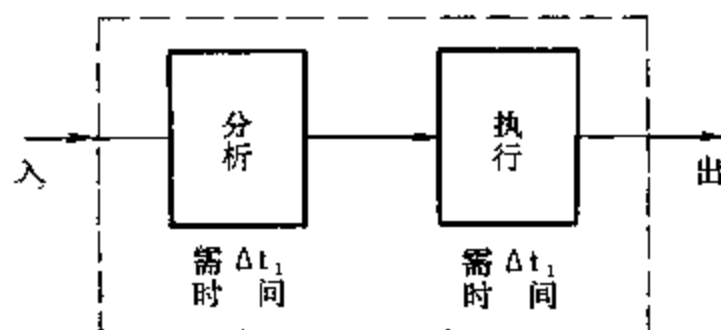


图 3.11 指令分解为“分析”与“执行”

转入“执行”子过程时，就可接收下一条指令进入“分析”子过程。若实现“分析”与“执行”子过程都需 Δt_1 时间，则就一条指令来看，是需要 $2\Delta t_1$ 才能完成；然而，从机器的输出来看，则是每隔 Δt_1 就能完成一条指令的解释。

这样，机器的吞吐率（这里指的是单位时间内机器所能处理的指令数或是机器能输出的结果数量）就由于把一条指令的解释过程分解成二个子过程而提高一倍。显然，若把“分析”子过程再分解成“取指令”、“指令译码”和“取操作数”子过程，并改进运算器的结构以加快其“执行”子过程，则可构成如图 3.12(a) 所示的解释过程。若这四个子过程都是需 Δt_2 完成，则指令解释的时(间)一空(间)图如图 3.12(b) 所示，图中的

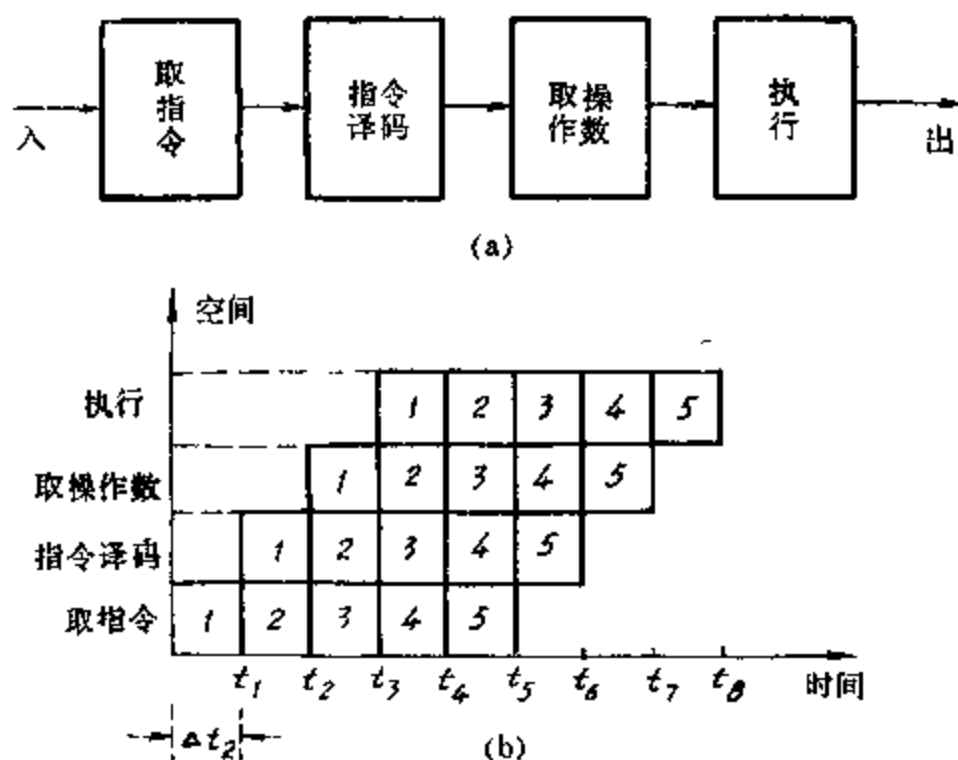


图 3.12 流水处理

(a) 指令解释的流水处理

(b) 流水处理的时(间)一空(间)图

1、2、3、4、5 表示处理机所处理的第 1、2、3、4、5、条指令。

看出，若完成一条指令的时间为 T ，则对分解为“分析”和“执行”二个子过程的，其

$$T = 2\Delta t_1$$

而对分解为“取指令”、“指令译码”、“取操作数”和“执行”四个子过程的，其

$$T = 4\Delta t_2$$

这样，虽然完成一条指令所需时间仍是 T ，但对图 3.11 是每隔 $\Delta t_1 = T/2$ 就可由处理机“流出”一个结果，即吞吐率提高了一倍；而对图 3.12 是每隔 $\Delta t_2 = T/4$ “流出”一个结果，即吞吐率比顺序方式提高了三倍。看出，此种工作方式和工厂中流水线的概念是相同的，那里也是每隔 Δt_i 由流水线流出一个产品，而制造这个产品的总时间却可能比 Δt_i 大得多。“流水方式”这个术语就是借用了流水生产线的名词。

显然，上节讲的“重迭”和这节讲的“流水”在概念上是密切联系的。可以这样看：“一次重迭”和“流水”的差别在于前者是把一条指令的解释过程只分解为二个子过程，而后者则是分解成更多个子过程。一次重迭可同时解释二条指令，而如图 3.12 的流水则可同时解释四条指令（1、2、3、4 或 2、3、4、5、）。

还要注意到，当流水线正常流动时，是每隔 Δt_i 就会流出一个结果；然而，在指令刚开始流动时，情况并不如此，由图 3.12 可看出，在 t_4 之前（即首条指令流入后的 $4\Delta t_2$ 时间内）流水线并没流出任何结果。也就是说，对指令流的首条指令来讲，流水方式和顺序方式是一样的。

2.1-2 流水结构的分类

流水结构有多种类型和不同的分类。我们从图 3.12 的流水结构开始分析，它可以向上

和向下扩展。

向下扩展指的是进一步分解成更多个子过程。由于吞吐量是反比于 Δt_i ，而 Δt_i 又是反比于子过程的个数；所以，若能分解成能流水进行的更多个子过程，而且各个子过程的完成时间大致相等，那吞吐率就会进一步增大。

图 3.12 的流水结构是把图 3.11 的“分析”子过程再分成“取指令”、“指令译码”和“取操作数”三个子过程；下面要讲到，它还可以进一步细分。当然，“执行”子过程也可细分，例如，把浮点加法运算分解成能流水进行的“求阶差”、“对阶”、“尾数相加”和“规格化”子过程就是一例。不同机器指令的“执行”子过程的复杂性不同，所需时间也不同。由于算术运算指令的“执行”子过程最复杂、最费时间，因此，对“执行”子过程流水实现的研究，其实就是对算术运算如何流水实现的研究。除了浮点加、减运算外，乘、除法运算也是适应于流水实现。例如，若相乘过程可分解成四个子过程（可以采用计算机原理课中讲过的多位一乘的方法），每个子过程分别由各自的加法器或是保留进位加法器实现，则流水乘法器的乘积流出率（乘法器对于一串相乘指令，在单位时间所能流出的乘积个数）要比非流水方式的高得多。

下面在第五章还要讲到，流水技术还可“向下”应用于对 Cache 存贮器和多体交叉主存的访问，使存贮器的频宽得以提高。

以 Amdahl 470V/6 的流水结构为例，看看“分析”子过程可以怎样细分。这台机器是和 IBM370 兼容的，它和上一节讲的 DJS-240 相似，其中央处理机分成指令分析部件（简称 I 部件）和执行部件（简称 E 部件，作用同运算器）。此外，还有通道（C 部件）、主存控制器（简称存控，S 部件）和高速 Cache 存贮器（简称 HSB，存指令和数据）。

I 部件采用流水结构。顺序的指令予取到 HSB，而后再由“指令选择逻辑”取入 I 部件。I 部件的主要功能除取指令外，为指令译码、访问通用寄存器（取操作数、基址值和变址值等）、计算访存操作数地址、取操作数、把操作数送往 E 部件并启动它、对 E 部件的运算结果进行校验和写回运算结果等。对于典型的需访存取一个操作数的指令，在 I 部件流水结构内，除取指令外，需执行的操作有以下九个，每个至少需一拍。

指令译码	D
访问通用寄存器取基址、变址值	R
形成访存操作数地址	A
启动取操作数	B1
从 HSB、通用寄存器读操作数	B2
把操作数送 E 部件并启动它	E1
E 部件结束运算	E2
对 E 部件的运算结果进行校验	C
把运算结果写回	W

若进流水线的指令都是上述那种类型的，则 9 个操作在流水线内的重迭状况如图 3.13 所示。若要取出的访存操作数还不在于 HSB，而需访主存才能取得，则 B1 和 B2 之间不是一拍，而是要等到能读入操作数时才执行 B2。同样，若运算复杂，则 E1、E2 之间也不是一拍，而是要等 E 部件结束运算时，才执行 E2。

看出，虽然 470V/6 的上述典型“分析”过程有 9 个操作，但实质上它至多只相当于细

指令序列	
第K+1条	D R A B1 B2 E1 E2 C W
第K+2条	D R A B1 B2 E1 E2 C W
第K+3条	D R A B1 B2 E1 E2 C W
第K+4条	D R A B1 B2 E1 E2 C W
第K+5条	D R A B1 B2 E1 E2 C W
第K+6条	D R A B1 B2 E1 ...
	第i第i第i第i第i第i第i第i第i第i第i第i第i第i第i第i + + + + + + + + + + + + + + + + 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 拍拍拍拍拍拍拍拍拍拍拍拍拍拍拍拍拍拍

图 3.13 Amdahl 470 V/6 I 部件的流水处理

分成 D、R、A、B、E、C 和 W 七个子过程。此外，还由于流水中相邻 D 之间安排成至少需隔 2 拍，所以对图 3.13 情况是同时解释 5 条指令。这种安排是为了便于解决流水处理中的转移问题。

上面讲的是解释指令的流水处理，而从图 3.12 的向上扩展可理解为由二个以上的处理机串行地对数据集进行处理，如图 3.14 所示。就是说，若进来的数据集需经多个任务的处理，且安排成各个任务分别由不同的处理机处理，则由于各个处理机都能同时工作，因此能流水地对不同的数据集进行处理，从而使处理能力得以有较大提高。

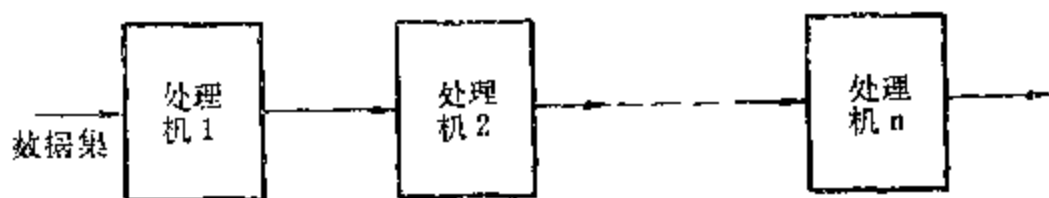


图 3.14 处理机间的流水处理

当然，在每个处理机内部，还可采用相应的流水技术。

综上所述，流水处理技术可用于部件级（如运算器内的浮点运算和乘、除运算等）、处理机级（如上述 Amdahl 470V/6 例子）和处理机间。

流水结构还可分成“动态”和“静态”以及“单功能”和“多功能”。“单功能”指的是只能实现一种功能（如只能实现浮点加、减法或只能实现乘法等）的流水处理；“多功能”则是指的同一个流水线可以有多种联结方式以实现多种功能。例如，ASC 的运算器流水结构就是多功能的。它有八个可并行工作的独立功能块，如图 3.15(a) 所示；当要进行浮点加、减法运算时，各功能块的联结如图 3.15(b) 所示；当要进行定点乘法运算时，各功能块的联结则如图 3.15(c) 所示。

“静态流水结构”指的是在同一时间内，上述多功能结构中的所有功能块只能按同一种运算的联结方式工作。例如，上述 ASC 的 8 个功能块只能或是都按浮点加、减运算工作，或是都按定点相乘运算工作。这样，只当进入的是一串具有相同运算的指令时，流水的效率才得以发挥，使各个功能块能并行地对多条指令的数据流水处理。然而，若进入的是不同指令相间的一串指令，例如是浮加、定乘、浮加、定乘……的一串指令，则上述静态流水结构的效率会下降到和顺序方式的一样。这是因为，当“定乘”指令的数据进入静态流水运算器时，运算器是按图 3.15(c) 联结；可是当数据经由“输入”块进入“相乘”块并继而往下流

时, 虽然输入、减阶、对阶移位、相加、规格化等功能块都是空闲的, 但流水运算器并不能

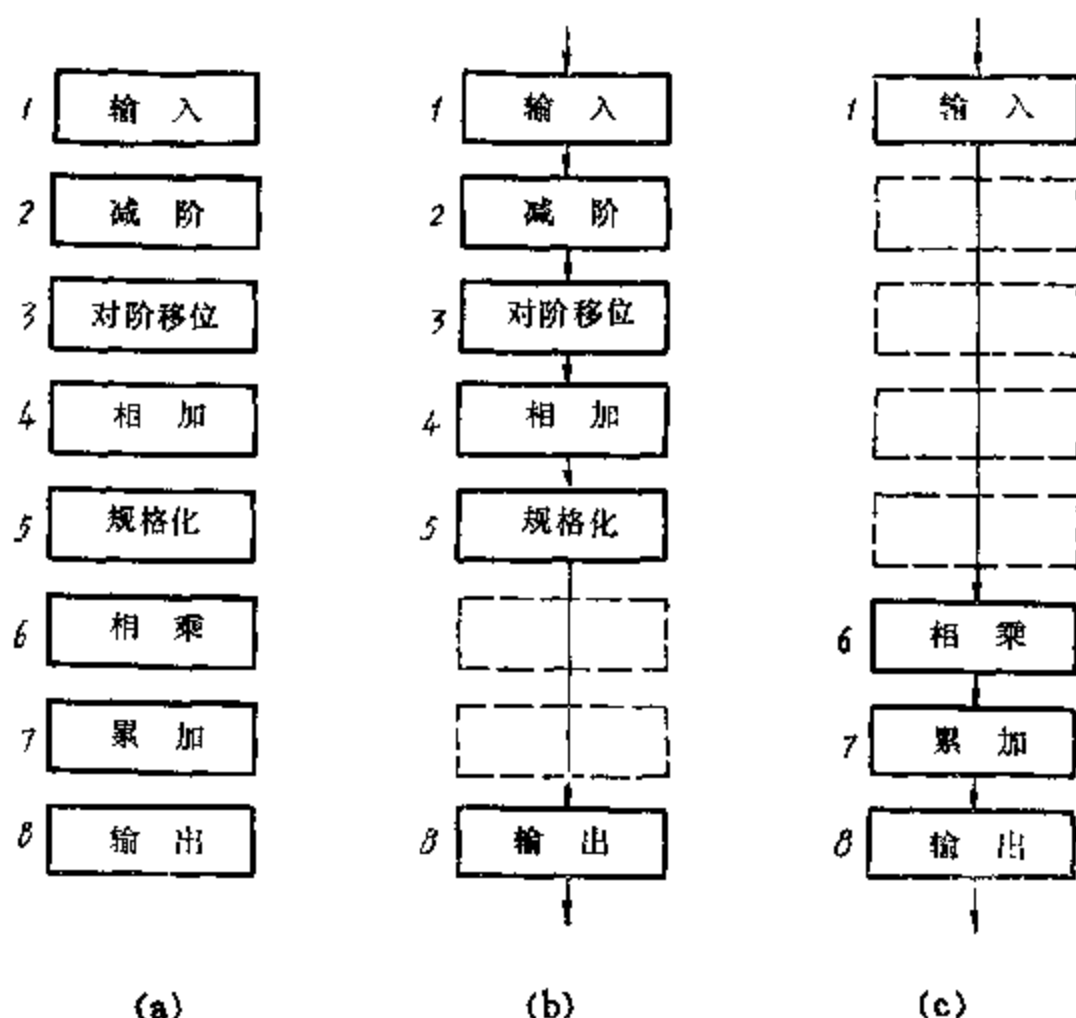


图 3.15 (a) ASC 多功能流水运算器的功能块
(b) 浮点加、减法运算时的联结
(c) 定点乘法运算时的联结

接着接收“浮加”指令的数据, 而是要等到相乘运算全部结束, 流水运算器切换到图 3.15(b) 之后, 才能开始“浮加”运算。目前, 绝大多数的流水结构都是上述静态式的, 这样, 就得要求程序设计者 (或是编译程序) 编制出 (或形成) 的程序应尽可能有更多个具有相同运算的指令串, 只有这样, 才能使得流水结构中的各个功能块能高效率地、满负荷工作。关于效率问题, 我们在下一节还要分析。

“动态流水结构”则是指的在同一时间内, 上述多功能结构中的各个功能块可按不同运算的联结方式工作。例如, 图 3.15 的各个功能块可以做到在同一个时间内, 当某些功能块正在实现某种运算 (如定乘) 时, 另一些功能块却在实现另一种运算。这样, 就不是非得是相同运算的一串指令才能流水处理。显然, 这对提高流水的吞吐率很有好处, 然而却因为在同一个时间内, 流水结构需能执行不同的多种运算, 从而使流水的控制要复杂得多。

此外, 流水机器还可分为标量流水机和向量流水机。向量流水机指的是机器具有第二章 § 1.3-2 所述的向量数据表示, 而且机器设置了相应的硬件, 使得能流水地对向量的各个元素并行处理; 标量流水机没有向量数据表示, 其流水状况前面讲过了。向量流水机是向量数据表示和流水技术的结合, 光有向量数据表示而没有相应的流水结构, 只能缩短程序的长度而不会使解题速度有大的提高; 同样, 光有流水结构而不采用向量数据表示, 则流水的效率也难以充分发挥。在以后, 我们还要讲述向量的流水处理。

综上所述, 流水技术的应用范围是广泛的, 方式是多样的。流水机器是单指令流, 因为指令是一条一条地由主存或 Cache 存贮器取进处理机进行解释, 只不过流水方式比起顺序方式或上一节所讲重迭方式进入处理机的各条指令间的时间间隔要短得多。同理, 流水机器也

是单数据流。虽然流水机的分析器或运算器可能有多个硬件功能块，然而，就每个子过程来讲，只有一个功能块与之对应。这样，流水机器当然不是多处理机系统，而是复杂化的单指令流、单数据流（SISD）系统。所以，我们不把它放在“多机系统”讲。当然，在多处理机系统中是可以采用流水技术以提高每个处理机的处理能力和速度的。

2.2 主要性能及其分析

本节讲述流水结构的吞吐率和效率（设备利用率）并结合具体实例进行分析。

2.2-1 吞吐率

吞吐率的定义在本章 § 2.1-1 已讲过，它是估量计算机系统性能好坏的一个重要标志。软件和硬件的性能都会影响到吞吐率的大小。就硬件来看，前面讲过，采用流水技术对提高吞吐率有很大作用。

结合图 3.12，设取指令、指令译码、取操作数和执行这四个子过程所需时间分别为 Δt_1 、 Δt_2 、 Δt_3 、 Δt_4 。这四个 Δt_i 都相等的情况在前面已分析过了；若它们不等，则处理机的吞吐率为：

$$\frac{1}{\text{MAX}\{\Delta t_1, \Delta t_2, \Delta t_3, \Delta t_4\}}$$

显然，它取决于流水线中最慢子过程所需的时间，这个子过程就是所谓的“瓶颈”。若

$$\Delta t_2 = 3\Delta t_1 = 3\Delta t_3 = 3\Delta t_4 = 3\Delta t_0$$

如图 3.16(a) 所示，则瓶颈在“2”。解决的办法可以有子过程再细分或采用多套设备并行的办法，如图 3.16(b)、(c) 所示。(b) 法在 § 2.1-1 已讲过；当然，并不是所有子过程都能

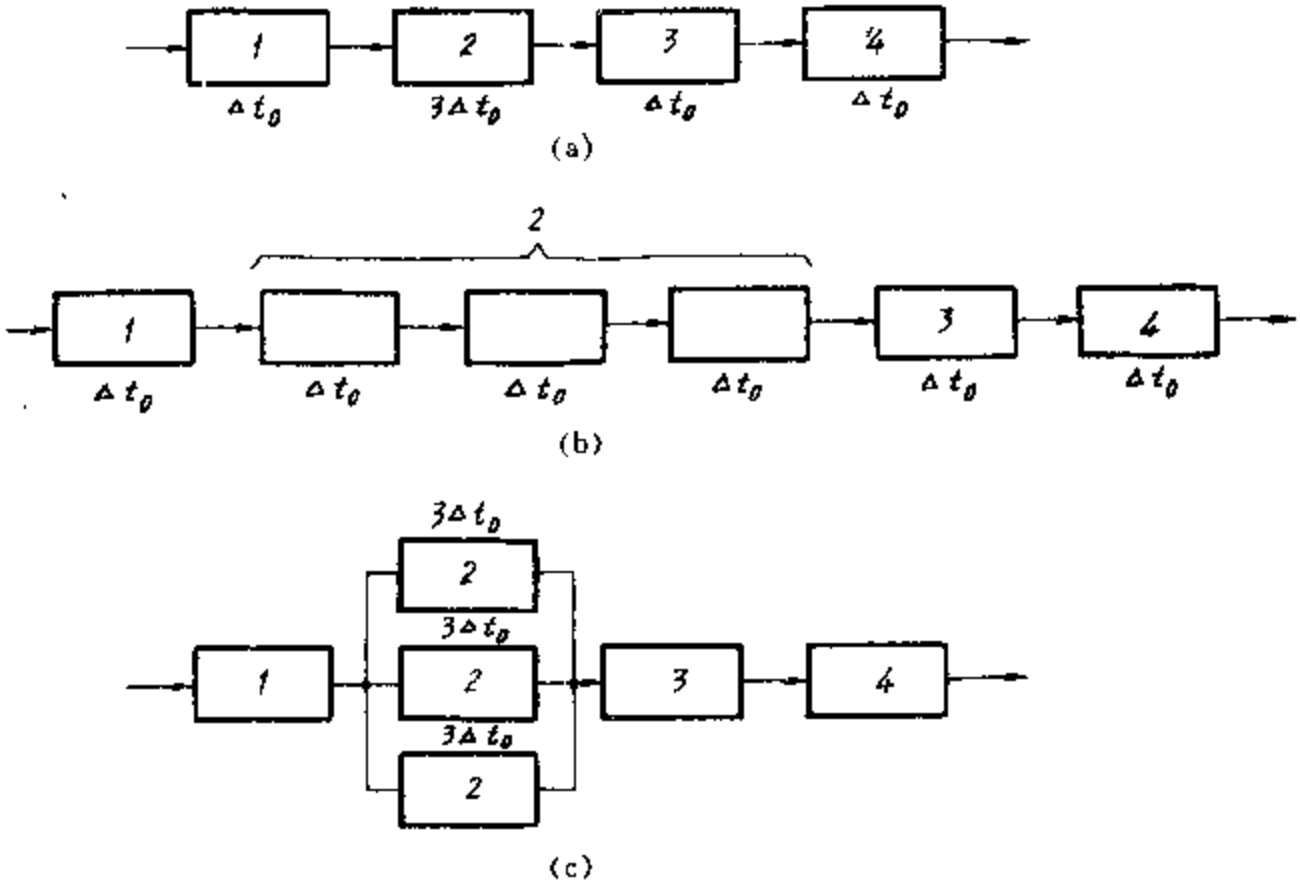


图 3.16 提高吞吐率的办法

- (a) 瓶颈在“2”
- (b) 用再细分子过程提高吞吐率
- (c) 用设备并联方法提高吞吐率

或适于再细分。若“2”子过程不能再细分，则可用(c)法，然而，它在任务分配（在各并行设备之间）和同步上，要比(b)法复杂。

上面讲的是流水线在连续流动时的最大吞吐率；然而，在实际运行中，由于程序构成和流水结构的限制，它几乎是不可能达到的。对于静态单功能流水运算器，只是对具有相同运算的一串指令，才能获得这个最大吞吐率，实际情况当然绝不会如此，而多是如图 3.17 所示的流一段时间、停一段 ΔT 时间，而后再流。

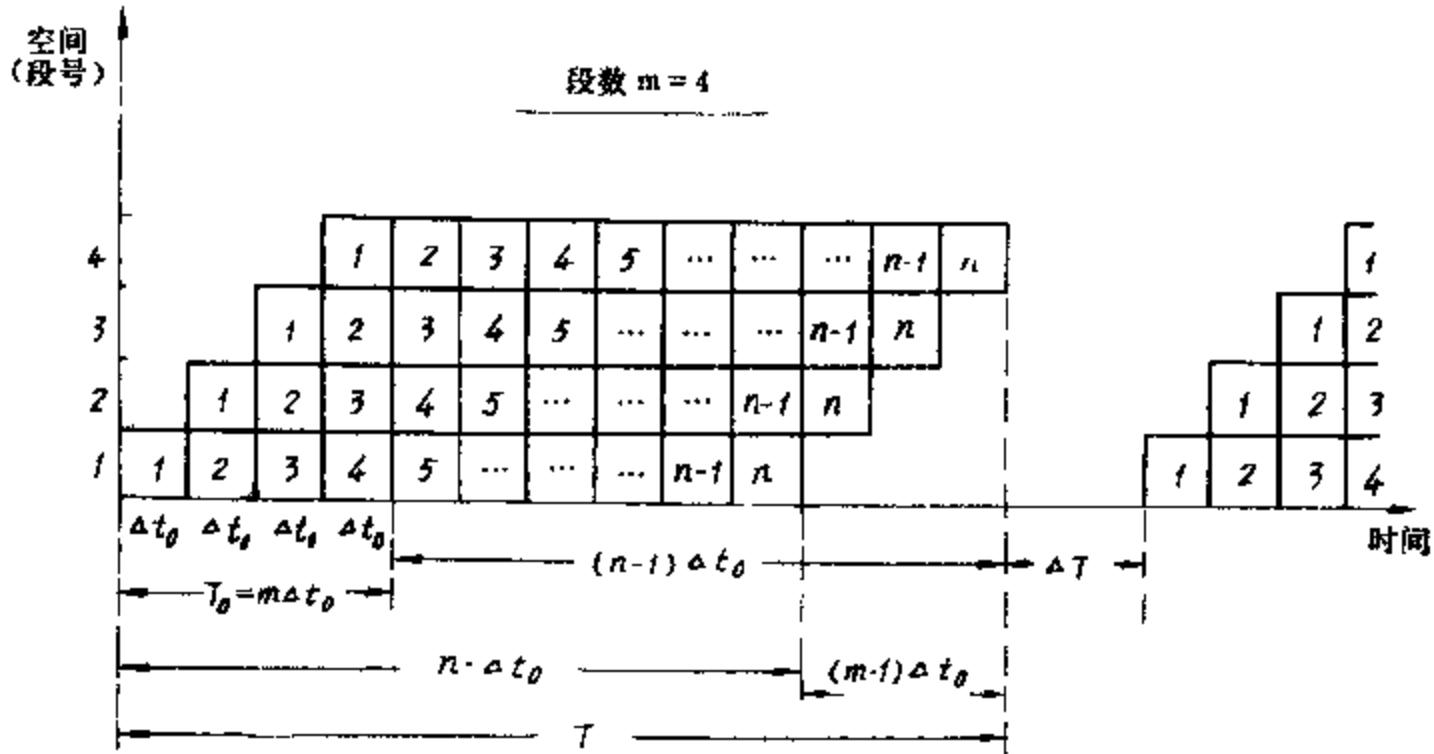


图 3.17 从时空图分析吞吐率

例如，对于单功能的浮点加、减流水运算器，当连续的 n 条浮点加、减运算指令流入时，其时空图如图 3.17 T 时间片那段所示。设浮点加、减操作分解成执行时间都是 Δt_0 的 4 个子过程，即完成一次浮点加、减操作所需时间 $T_0 = 4\Delta t_0$ 。这样，执行完 n 次浮加操作，由流水线输出端获得 n 个运算结果所需时间为

$$T = n \cdot \Delta t_0 + (m-1) \cdot \Delta t_0$$

m 为子过程的个数（即流水线内有 m 个可并行执行的功能块或段），这里是 $m=4$ 。这样，流水运算器的吞吐率 TP 为：

$$TP = \frac{n}{n \cdot \Delta t_0 + (m-1) \cdot \Delta t_0} = \frac{1}{\Delta t_0 \left(1 + \frac{m-1}{n}\right)}$$

看出，只当 $m \ll n$ 时，吞吐率才能接近于 $\frac{1}{\Delta t_0}$ ，即接近于流水线的最大吞吐率。机器手册给的吞吐率一般就是这个值。显然，若 m 值大，而流水线能够连续流动的概率并不高（这是由于目的程序中相同操作的指令串少，或是由于存贮系统来不及提供为连续流动所需的操作数）， n 值不够大，那流水线的实际吞吐率就要比手册给的小得多。

2.2-2 效率

由图 3.17 可看出，并不是流水线内的各个段（功能块）都一直满负荷工作。因此，需要分析流水线所用设备的效率（或曰设备利用率），从中寻找提高流水线性能的途径。

我们先对图 3.17 时空图 T 时间片那段的流水线效率进行分析。看出，对第 1 段（第 1

个功能块) 其效率 η_i 为:

$$\eta_i = \frac{n \cdot \Delta t_0}{T} = \frac{n \cdot \Delta t_0}{m \cdot \Delta t_0 + (n-1) \Delta t_0} = \frac{n}{m + (n-1)}$$

对这个例子, 各段的效率都是此值, 即

$$\eta_i = \eta_1 = \eta_2 = \eta_3 = \eta_4 = \frac{n}{m + (n-1)}$$

若 $m=4$, $n=10$, 则 $\eta_i=0.77$; 若 $m=4$, $n=1000$, 则 $\eta_i=0.997$ 。看出, 在 $n \gg m$ 时, $\eta_i \approx 1$ 。至于整个流水线的效率 η , 若 m 个段的 Δt_i 都相等, 则

$$\eta = \frac{m \cdot (n \cdot \Delta t_0)}{m \cdot T} = \eta_i$$

就是说, 对于图 3.17 所示, 在执行每个任务 (指令、操作) 时, 各段都只流过一次的所谓线性流水线, 若 Δt_i 都相等, 则整个流水线的效率等于各段的效率。

然而, 就线性流水线来说, 若各段的 Δt 不等, 而且设备量也不等 (例如, 对浮点加、减流水运算器, “减阶”段的设备量要比采用全移位器、每次只需一次对阶移位的“对阶”段的设备量少得多), 则应给各个段赋予不同的“权” (α_i), 参看上述 η_i 表示式可引出线性流水线效率的普遍式为:

$$\eta = \frac{n \left(\sum_{i=1}^m \alpha_i \cdot \Delta t_i \right)}{\sum_{i=1}^m \alpha_i \left[\sum_{j=1}^m \Delta t_j + (n-1) \Delta t_j \right]}$$

Δt_j 为瓶颈段的执行时间。

对于不是线性的流水线, 例如对浮点加、减流水运算器, 若“对阶”段不是采用全移位器, 则每执行一个任务, “对阶”段可能需多次循环, 而“减阶”段则只需通过一次; 对于这种非线性流水线, 其 η 的表示式当然要比上式复杂。然而, 只要能画出其时——空图, 我们就可用下式求出其 η 值:

$$\eta = \frac{n \text{ 个任务的总的加权时——空区}}{m \text{ 个段的总的加权时——空区}}$$

结合图 3.17, 分母为对应所选时间片的整个时——空区 (还要加权), 分子为对应所选时间片有标号的时——空区。显然, 时——空图中的空白区愈多, 流水线的效率就愈低。在以后的实例中, 还要讲述如何由具体时——空图求 η 值。

流水线的效率和吞吐率之间一般是正比关系。以图 3.17 对应 T 时间片的情况为例,

$$\eta = \frac{n}{m + (n-1)} = \frac{n}{n + (m-1)} = TP \cdot \Delta t_0$$

就是说, 为提高效率所采取的措施对提高吞吐率也有好处。这样, 为减少时——空图空白区所采取的措施也必然会对提高吞吐率有好处。

联系图 3.17, 减少 ΔT 空白区使流水线尽可能不间断地流动, 对提高效率当然有好处, 而这同样提高了吞吐率。对于多功能静态流水线, 就是在 $\Delta T=0$, 即一批任务本来是可以连续流入时, 但若其中要求有功能切换 (如由“浮加”切换到“定乘”), 联系图 3.15 的例子,

那在切换处仍然会出现如图 3.18(a)所示的空白区,使效率不能进一步提高,若多功能流水线具有动态结构,即在流水线内,可以有些段执行某种功能(如浮加),而同时另一些段执行另一种功能(如定乘),则切换处的空白区就可减少,如图 3.18(b)所示。由于输出段(第 8 段)是“浮加”和“定乘”都要用到的,因此,只能在“浮加”用完后,“定乘”才能开始用,由此就可定出图 3.18(b)的时——空图。

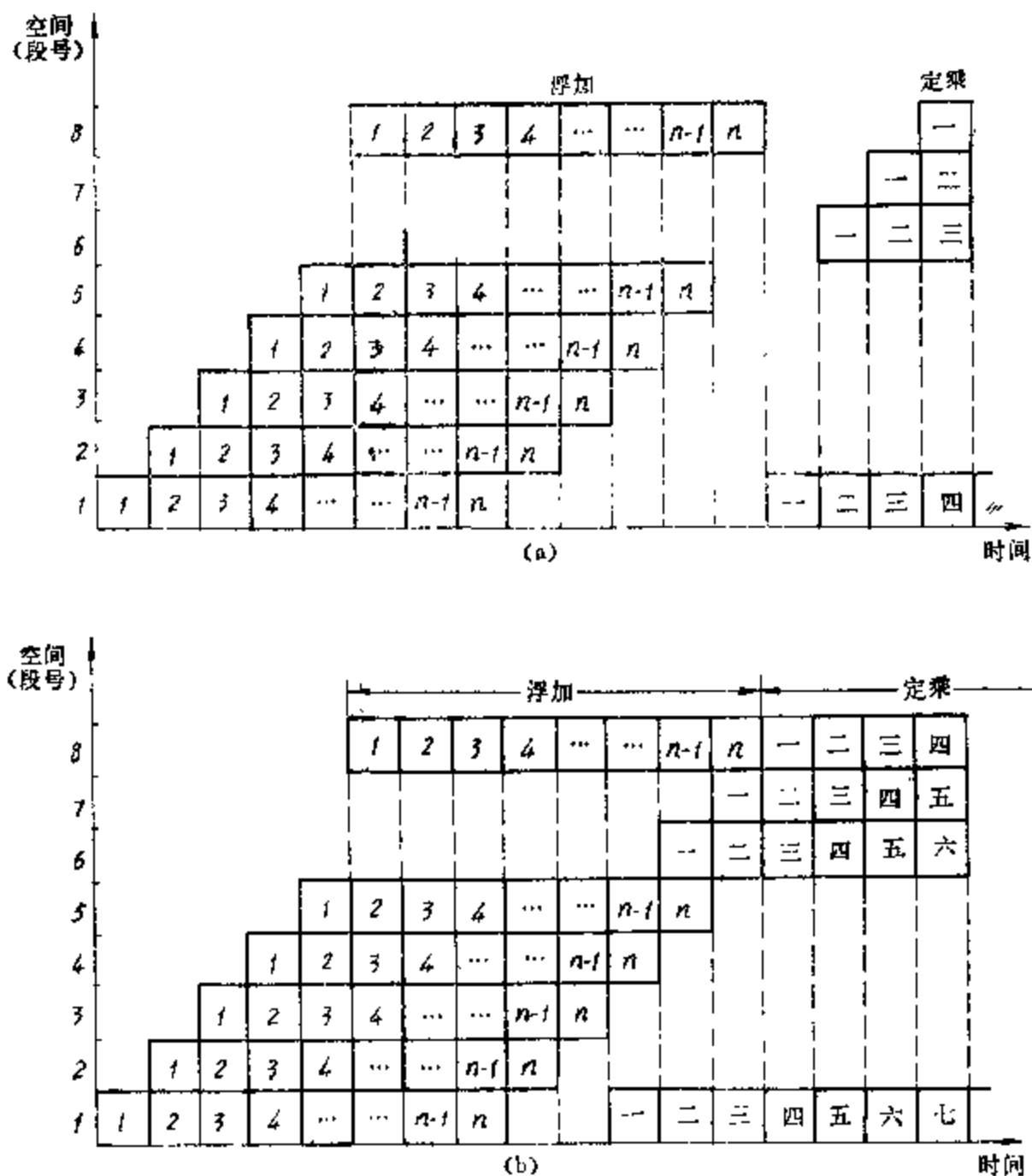


图 3.18 动、静态多功能流水线时——空图举例
(a) 静态 (b) 动态

2.2-3 实例分析

现在以浮点加法流水运算器执行

$$Z = A + B + C + D + E + F + G + H$$

为例,画出其时——空图,分析其吞吐率和效率。设流水运算器为线性的,分成“减阶”、“对阶”、“尾加”和“规格化”四段,如图 3.19(a)所示。由于此算式的各操作数没有先后次序的限制,所以其实现算法可灵活多样,其中一种为:

$$Z = [(A + B) + (C + D)] + [(E + F) + (G + H)]$$

它需要执行七次相加。若各段的执行时间都是 Δt_0 ,且“部份和”可以直接返回“加数”端或是暂存于相应寄存器,则实现此运算的时——空图及“被加数”、“加数”、“和”端的

变化如图 3.19(b)所示。阴影部分表示该段在工作。

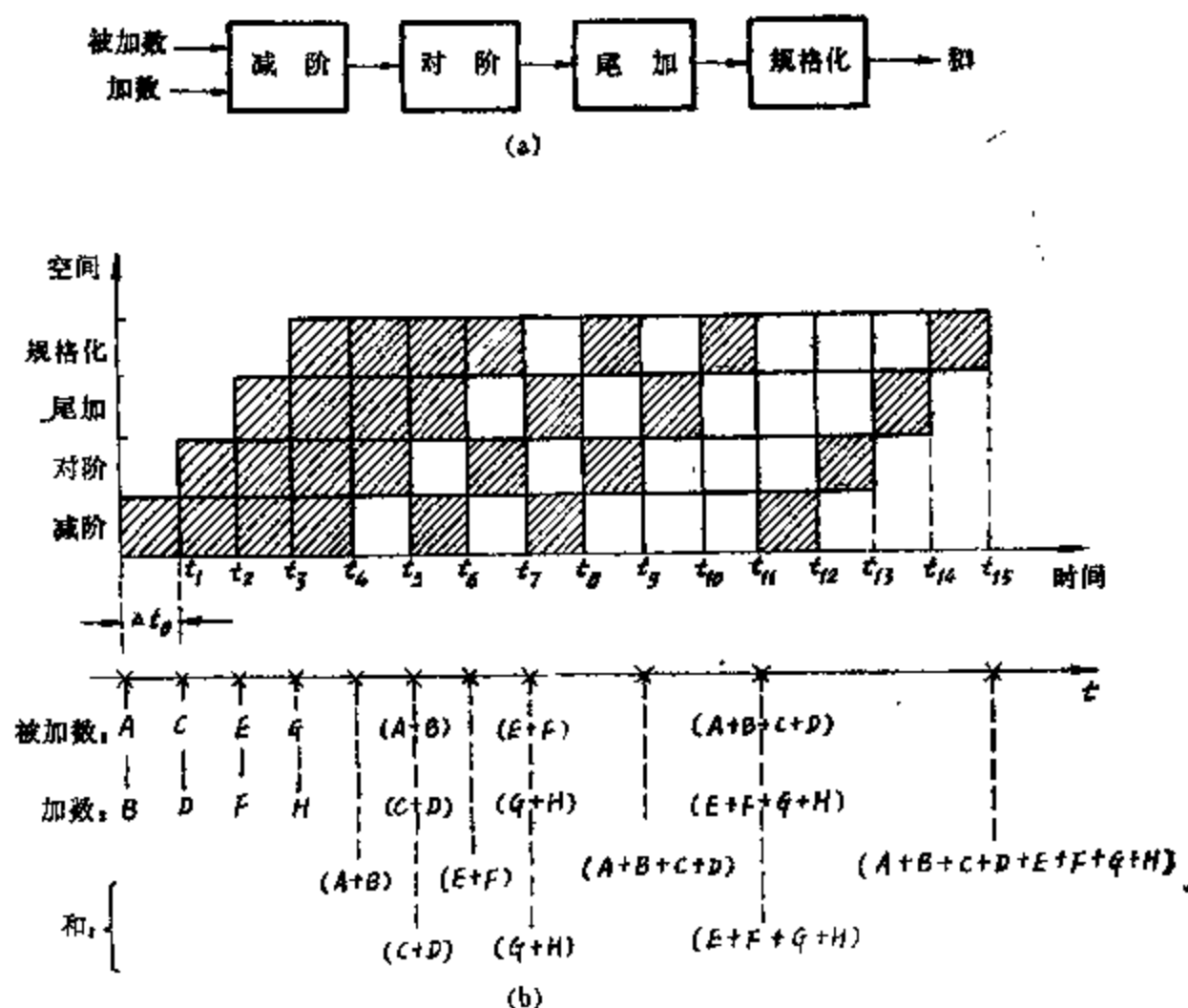


图 3.19 实现 $Z = A + B + C + D + E + F + G + H$ 运算

(a) 浮加流水线

(b) 时——空图及输入、输出的变化

看出，需化 $15\Delta t_0$ 才能求得 Z 。就求这个 Z 来讲，流水运算器的效率可由阴影区与 4 个段的总时——空区的比值求得，由图可看出，

$$\eta = \frac{4 \times 7}{4 \times 15} = \frac{7}{15} \approx 0.47$$

而其吞吐率可由在 $15\Delta t_0$ 时间片内共输出多少个“和”求得，由图可见，共输出 7 个“和”，即

$$TP = \frac{7}{15 \cdot \Delta t_0}$$

若不是流水运算器，则求 1 个“和”需 $4\Delta t_0$ ，求 7 个“和”需 $28\Delta t_0$ 。虽然这个流水运算器在求 Z 时的效率连 50% 都没到，然而其解题时间 ($15\Delta t_0$) 仍比非流水运算器的解题时间 ($28\Delta t_0$) 短得多。当然，浮加流水运算器要求“对阶”和“规格化”能同时进行，所以需有二套全移位器，但设备量很贵的尾数加法器却只需一套，因此它的设备量定比二套非流水运算器的少。

用流水运算器求 Z 值之所以效率不高是由于需把它的输出回授到输入，致使不是每拍都能有数进入。由此例看出，流水运算器最适合于解具有同一操作类型，而输出与输入之间又无任何联系、相关的一串运算。向量运算，如求

$$a_i = b_i + c_i \quad i = 0, 1, 2, \dots, n$$

就是满足这种要求的一例。只要 b_i 、 c_i 能连续不断地取得，流水运算器就能连续不断地流

动, 若 n 值很大, 则效率可接近于“1”, 每隔一个 Δt_i 就可输出一个 a_i 。

2.3 相关处理和控制机构

上面讲过, 只有使流水线连续不断地流动, 即不出现“断流”情况, 才能获得高效率。造成“断流”的主要原因除了编译形成的目的程序不能发挥流水结构的作用和存贮系统供不上为连续流动所需的指令和操作数以外, 就是由于出现了“相关”。在本章的 § 1.2 和 § 1.3 已讲过了重迭机器的指令相关、主存操作数相关和通用寄存器的操作数相关及变址值相关。显然, 这些相关, 在流水机器中也都仍然会出现; 而且, 由于流水是同时解释更多条指令, 所以相关状况要比重迭机器的复杂。如果处理不当, 必会使流水效率显著下降。

此外, 流水机器在遇到转移指令, 尤其是条件转移指令时, 其效率会下降。例如, 当流水机器的转移条件码是由条件转移指令本身形成, 或是由它前一条的指令形成 (大多数程序就是如此), 那只有当它流出流水线时, 才能建立转移条件, 并依它决定下条指令地址, 那么从条件转移指令进入流水线, 译码出它是条件转移指令直至它流出的整个期间, 流水线就不能继续往前处理。而且, 若转移成功, 且转移出指令缓冲器, 那就要从由主存取指令开始重新流动。这些当然会使流水机器的效率和吞吐率严重下降。按照我们在 § 1.2 讲的“相关”定义, 由于转移指令和它之后的指令之间存在上述关联, 从而不能同时解释, 因此这也是相关的一种。我们可以称它为全局性相关, 而把前面讲的那些称之为局部性相关。全局性相关对流水效率的影响要比局部性的大。不处理好相关问题, 流水机器的实际吞吐率会严重下降。我们在设计机器或选购机器时, 都决不能只注意于连续流动时的效率和吞吐率。下面分别分析这二类相关的处理及所需的相应控制机构, 最后还一并简述流水机器的中断处理。

2.3-1 局部性相关的处理

包括指令相关、访存操作数相关和通用寄存器相关等这些局部性相关, 都是由于在机器同时解释的多条指令之间出现了对同一个单元 (包括主存单元和通用寄存器) 的“先写后读”要求。因此, 如 § 1.2, § 1.3 已讲过的, 重迭机器在处理这些局部性相关时可以有二种办法: 一种是推后对相关单元的读, 直至写入完成; 二是设置相关专用通路, 使得不必先把运算结果写入相关存贮单元, 再由它读出后才能使用, 而是经相关专用通路直接使用运算结果; 这样, 就可省去“写入”和“读出”那二个存贮周期。

由于流水技术是重迭技术的发展, 因此上述这二种方法的思路也适用于流水机器处理局部性相关, 例如 STAR-100 的相关直接通路就是如此; 但由于流水机器是同时解释更多条指令, 所以其相关直接通路的设计当然要比重迭的复杂。

对流水的流动顺序的安排和控制可以有二种思路。一种是使流水线输出端的任务 (指令) 流出顺序和输入端的流入顺序一样。这样, 指令是一条跟着一条在流水线中流动, 如图 3.20 所示, 第 j 条跟着第 i 条、第 i 条又跟着第 h 条等等。一旦第 j 条指令进入第 2 段 (例如是读操作数段) 时, 判出和它之前的某条指令 (例如当时是在第 4 段的第 h 条指令) 有操作数相关, 那就使第 j 条指令“停住”在第 2 段, 直至第 h 条指令达到第 7 段 (写段), 完成写入存贮单元后, 在第 2 段的第 j 条指令才执行读操作数, 并继续往前流动。显然, 这里的“停住”和前面 § 1.3 讲的“推后”在概念上是相同的。

对于要求顺序流动的流水线, 在第 j 条指令停住在第 2 段时, 它之后的指令 (如第 k 条)

指令地址:	n	j	m	l	k	i	h	(可以不顺序流动的)
	k	j	空	空	空	i	h	(顺序流动的)
	k	j	i	h				(判出 j、h 相关)

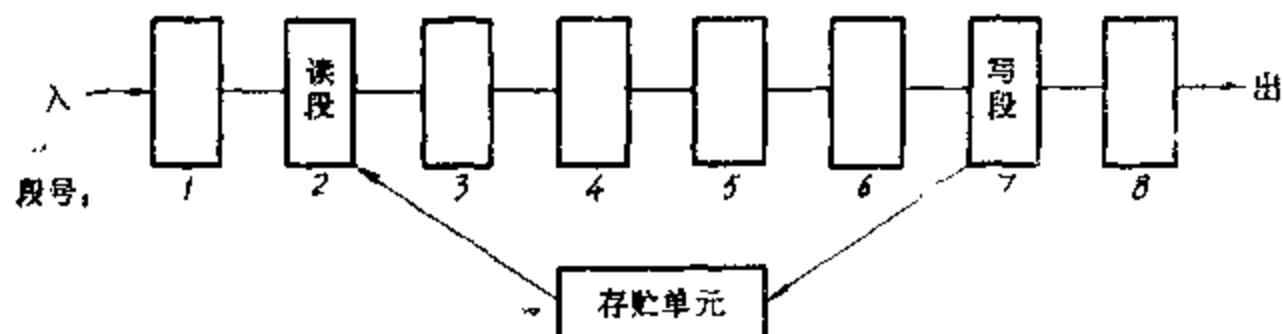


图 3.20 用“停住”法处理操作数相关

也必须停住，致使流水线的输入端也停止接收。因此，在第 h 、 i 条指令继续往前流时，第 3、4、5 段会逐步空出没用，如图 3.20 “顺序流动的”那行所示。这显然会降低流水线的效率和吞吐率。然而，采用顺序流水的方法，控制比较简单，ASC 机就是采用这种方法。

如果第 j 条指令之后的指令，如 n 、 m 、 l 、 k 和它之前的指令都没有相关（这在实际程序中是会经常遇到的），那为什么不使它们越过第 j 条指令继续往前流动，从而使流水线的效率和吞吐率得以比顺序流动的高。这就引出了流动顺序的安排和控制的第二种思路，即流水线输出端的指令流出顺序可以和输入端的流入顺序不一样。这样，指令在流水线内的流动顺序就可能如图 3.20 的“可以不顺序流动的”那行所示。由图看出，这使流水线的效率和吞吐率都提高了，但指令的流出顺序却成为 h 、 i 、 k 、 l 、 m 、 j 、 n ，与流入的 h 、 i 、 j 、 k 、 l 、 m 、 n 不一样，这当然会增加控制的复杂性。既然这是异步流动方式，那后续指令当然可以不止是越过因相关而需停住的指令，而且还可越过执行时间（即所需用到的段数）比它们长的指令。

异步流动会使相关状况复杂化。例如，若第 k 、 i 条指令都有写操作，且是写入同一个存贮单元，那该单元的最后内容本应是第 k 条的写入结果；然而，或是由于第 k 条指令的执行时间短得多，或是由于第 i 条指令执行时间很长或有“先写后读”相关，就会出现第 k 条指令先于第 i 条达到“写段”的情况，从而使得该单元的最后内容错为第 i 条的写入结果。我们称这种情况为出现“写——写”相关。同理，若第 i 条指令的读操作和第 k 条指令的写操作是对应同一个单元，那第 i 条指令读出的本应是该单元的原存内容；可是，若第 k 条指令越过第 i 条指令向前流，且其写操作在第 i 条指令的读操作开始前完成，那第 i 条指令就会错误地读出了第 k 条指令的写入结果。我们称这种情况为出现了“先读后写”相关。显然，它与“写——写”相关都是只在异步流动时才能出现。在设计这种方式的控制机构时要考虑到如何处理好这二种相关。

若在图 3.20 的第 7 段与第 2 段之间设置如 § 1.3 所讲的相关专用通路，那第 j 条指令就可提前二个存贮周期往前流动。由于流水机器是同时解释更多条指令，而且具有多个可并行工作的功能部件，当然不能在这些功能部件之间为各种局部性相关都设置相应的相关专用通路，而应由统一的总线结构进行处理。下面结合 IBM 360/91 浮点运算器的公共数据总线 (Common Data Bus, CDB) 结构和相应的相关处理来说明。

360/91 的运算器联结框图如图 3.21 所示。运算器的每个操作由浮点操作站 (FLOS) 的每行输出控制。每行指明操作内容和“源”及“目的”，如

ADD F2, FLB1 $(F2) + (FLB1) \rightarrow F2$

MD F2, FLB2 $(F2) * (FLB2) \rightarrow F2$

F2 表示是浮点寄存器 (FLR) 的第 2 号寄存器, FLB1 表示是浮点操作数缓冲器 (FLB) 的第 1 号寄存器。当然, 浮点操作站的每行并不是机器指令, 而是指令分析器在对机器指令进行分析后送往运算器的操作命令。

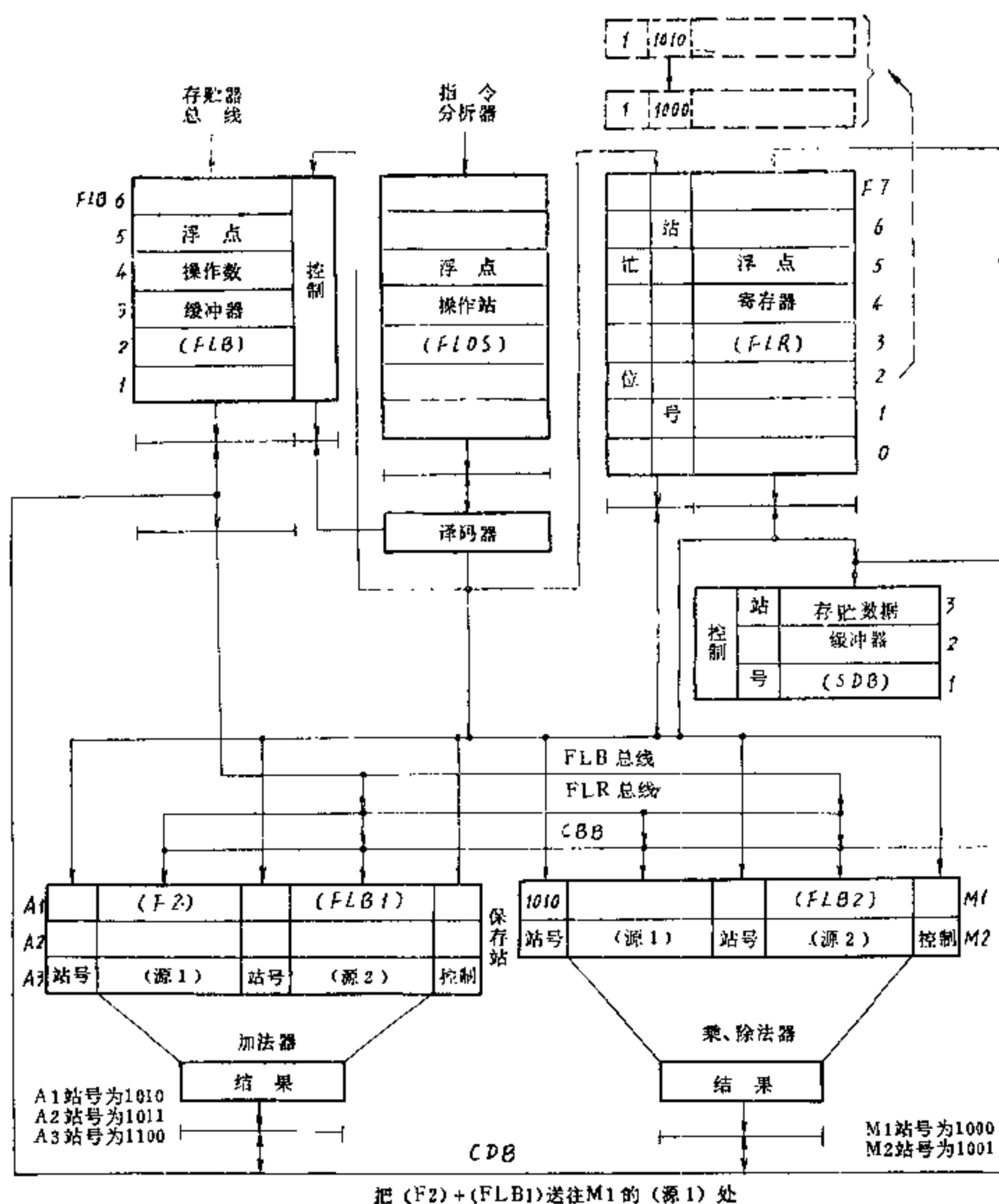


图 3.21 IBM360/91 的运算器联结框图

操作命令指明二个操作数的来源, 即源₁与源₂, 源₁还用于存结果, 即目的 = 源₁。一般源₁指明 FLR 的寄存器号, 所以中间运算结果是存入 FLR; 这样, 就可能出现浮点寄存器操作数相关, 如上面 ADD、MD 这二条操作命令就是如此。

IBM360/91 用“忙位”判相关, 用“站号”控制相关专用通路的联结。下面结合上述 ADD、MD 例子来说明。当 FLOS 送出

ADD F2,FLB1

操作命令时,它控制由FLR取得(F2),由FLB取得(FLB1)送往加法器保存站(设送往A1),并将F2的“忙位”置“1”以指明该寄存器的内容已送往保存站等待运算,且F2已成为“目的”寄存器,准备接收由加法器来的运算结果。这样,对“忙位”为“1”的 F_i ,其内容再不能被读出作操作数用。F2的“站号”是用于控制把运算结果经CDB送回F2。由于(F2)是送往A1,所以其站号置成A1(即“1010”),在加法器对A1的源₁,源₂进行运算后其运算结果是送往“站号”字段为A1的寄存器,即送回F2。送回后将F2的“忙位”和“站号”都置“0”。

若在F2的“忙位”已为“1”时,FLOS送出

MD F2, FLB2

操作命令,就出现了F2相关,那当然不能把此时的(F2)送往乘法器保存站,而应是把原存在F2的站号“A1”(指明F2应有内容的来源)送往M1的“源₁站号”,并把F2内的站号由A1(1010)改为M1(1000)以指明应改从M1接收运算结果,如图3.21右上方虚线框所示。这样,当加法器对A1站的(源₁),(源₂)进行相加,经CDB送出结果时,就不是送到F2,而是直接送进其“源₁站号”为A1的M1(这相当于相关直接通路)作(源₁)用;而乘法器是在M1的(源₁),(源₂)都已有内容时,才能对它执行相乘(这相当于推后相乘的执行),并把乘积经CDB送往其“站号”字段为M1的F2寄存器。

在加法器和乘、除法器输入端设置保存站的主要目的是使得这些运算部件可以在某个操作命令或因相关而需推后执行,或因执行时间过长(如相除)而还没完成时,仍能继续由FLOS接收操作命令,这当然是前述异步流动方式。

在360/91,控制相关处理的信息是随着每个操作命令与操作数一起流入保存站。这种分布式的控制方式大大简化了同时出现多个相关及多重相关的处理;它要比集中式的(如CDC-6600所用)灵活,且处理能力强。目前,大多数流水机器都是采用类似360/91的分布式控制方式。

综上所述,流水机器处理局部性相关的基本思路和重迭机器的一样,也是“推后执行”和“相关专用、直接通路”。

2.3-2 全局性相关的处理

前面讲过,流水机器一定要处理好转移型指令,尤其是条件转移所引起的全局性相关。下面分析一些常用的处理方法。

一、猜测法

为了在遇到条件转移指令后,流水线仍能继续往前流动,绝大多数机器都采用了所谓“猜测法”技术。就是说,在流水线译码出某条指令是条件转移,而在它所需的条件码还没建立前(一般,指令译码是在流水线始端,而条件码多要在流水线尾端才能建立),机器猜选转移分支中的一个,按它继续往前流,即继续解释“条件转移”指令后的那些指令。猜选哪个分支是固定的,可以是转移成功分支,也可以是转移不成功分支。IBM360/91是选的转移不成功分支,因为这可以使流水线按原来顺序继续往前流动,如图3.22所示,沿虚线流动继续解释第 $i+1$, $i+2$, $i+3$,……条指令。设第 i 条指令所用的条件码是在解释到第 $i+4$ 条指令时出现(这是由于流水机器能同时解释多条指令)。若条件码是对应于不成功转

移的，那就是猜对了，流水线沿虚线路径继续往前流动；若条件码是对应于成功转移的，那

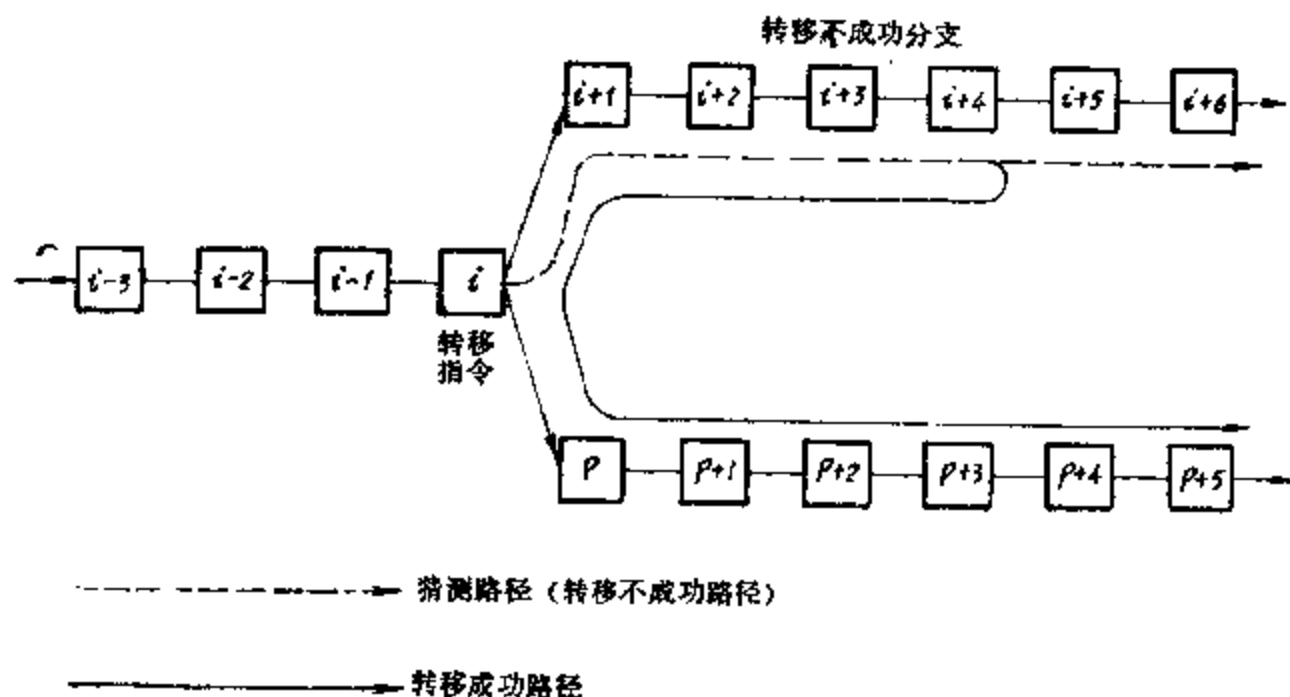


图 3.22 用猜测法处理条件转移

就是猜错，就要改为沿实线路径流动，报废已对第 $i+1$ 、 $i+2$ 、 $i+3$ 、 $i+4$ 条指令的解释，重新回复到分支点，并沿着转移成功分支解释第 P 、 $P+1$ 、 $P+2$ 、……条指令。

看出，控制机构的设计要保证在猜错而需返回到分支点时，能恢复分支点处的原有现场。对于图 3.22，还必需保证在返回前对第 $i+1$ 、 $i+2$ 、 $i+3$ 、 $i+4$ 条指令的已有解释全部报废，且不能影响原有现场的恢复。例如，若第 $i+2$ 条指令为

$$(R1) + (N) \rightarrow N$$

则一旦这条指令被解释到执行完，那原有现场就恢复不了，因为 N 单元的内容已被破坏了，使得已进行的解释报废不了。因此，当机器沿猜测路径对第 $i+1$ 、 $i+2$ 、 $i+3$ 、 $i+4$ 条指令进行解释时，一定不能破坏到存贮单元（寄存器）的原存内容。360/91 是安排成只进行译码和准备好操作数，但在还没能判定是否猜对，即在转移用条件码出现前，不执行运算；当然也可以安排成解释到运算完毕，但不送回运算结果，有的机器就是如此。

为了在猜错时能尽快地返回，转入另一分支，与沿猜测路径向前流动的同时，还可由存贮器予取转移成功分支的头几条指令（360/91 为予取二条双字长指令字）放在转移目标缓冲器，以便在猜错返回时，不必从访存取指令开始。

由于程序结构本身的特点，条件转移二个分支的出现概率是可能预估的。如果程序设计者或编译程序能把出现概率高的分支安排为猜选分支，那就会减少由于处理条件转移所引起的时间损失。

二、条件码的加快和提前形成

由以上的分析可看出，尽快、尽早获得条件码对流水机器简化条件转移的处理有好处。

就一条指令来讲，并不是非得在执行完毕，取得运算结果后，反映运算结果的条件码才能形成。例如，对相乘指令，反映其结果乘积是正、是负、是“0”的条件码是可以在获得操作数后，但在执行相乘运算之前判定。由于相乘（尤其是相除）操作所需时间长，这种安排对于使条件码提前形成是有好处的。Amdahl 470 V/6 就是如此，它在流水运算器输入端设置的 LUCK 部件能对大多数指令予判出它们的条件码，从而在具体运算前就能提前送条

件码到指令分析部件。

对一串指令来讲，也是有可能设法使条件码提前形成。例如，对下述FORTRAN程序，

```

SUM = 0.0
DO 20 I=1, N
  A = B(I) * C(I)
  IF(A, GE, 0.0) GOTO 15
  D = A * E(I)
  GOTO 5
15  D = A * F(I)
   5  G(I) = G(I) + H + D
20  SUM = SUM + G(I)
END

```

编译成的目的程序的框图如图 3.23 所示。图中标出每框的功能，只写出与分支有关的几条机器指令。LD 为取数；MUL 为相乘；BGE 为按“ ≥ 0 ”条件码转移；DEC 为减“1”；BE 为按“等于 0”条件码转移。RN 为循环次数计数器。

本例用了 BGE 和 BE 条件转移指令，试分析其条件码是否能提前形成。

由于第 2 框是求 A 乘积，而 BGE 又是按乘积是否 ≥ 0 转移，因此 BGE 只能紧挨着 MUL，如图上所示。这里，条件码的提前形成只能用上面讲过的，在相乘运算之前，予判乘积是否 ≥ 0 。

至于第 5 框，它是求 G(I) 和 SUM，而 BE 所用条件码却是和它们无关。只是因为现有机器是每条指令的执行都能改变之前的条件码状态，所以形成 BE 指令所用条件码的 DEC 指令必须与 BE 指令紧挨

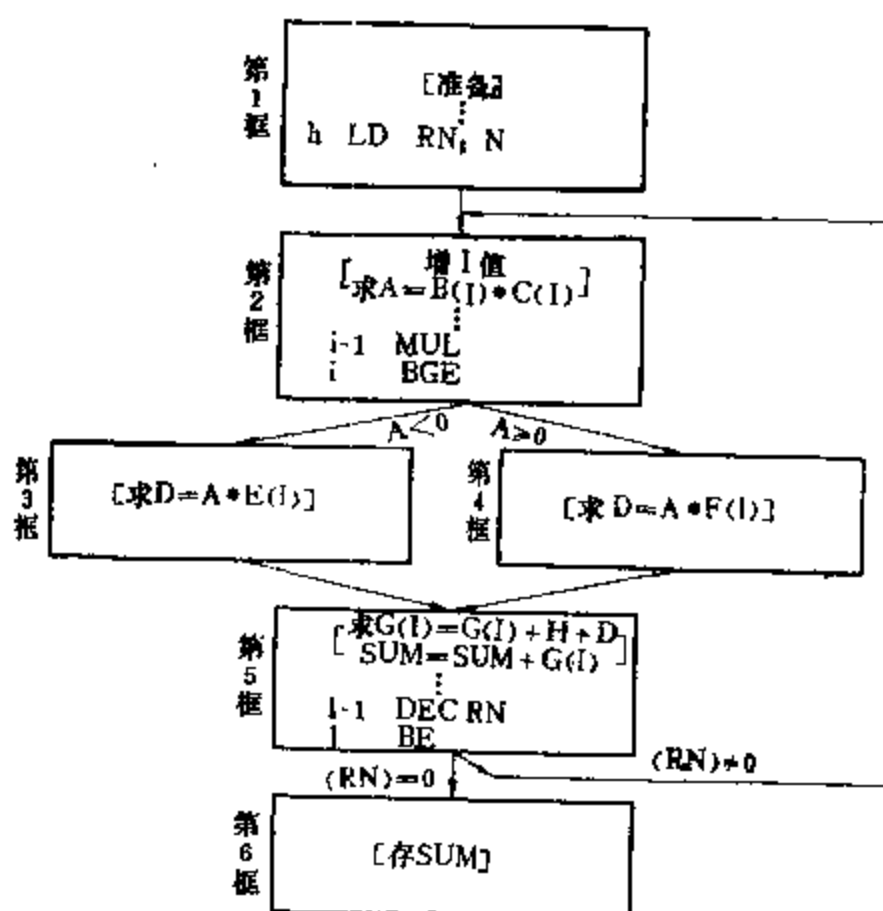


图 3.23 有分支的目的程序框图

着。显然，从执行顺序来看，完全可以把 DEC 指令提前执行(如放在第 5 框首)，使得 BE 所需条件码可以提前形成，从而控制机构能在解释第 5 框，求 G(I)、SUM 的同时，已能判定它之后的分支；这样，就有可能做到同时解释第 5 框和第 2 框，以利于流水线在遇到条件转移指令时，仍能连续流动。当然，为此需增设专用的条件码寄存器 CC_s ，它的存入和使用可由机器指令的相应特征位指明：

	特征位		
$l-n$	1	DEC RN	$CC \rightarrow CC_s$
\vdots	\vdots	\vdots	
l	1	BE	按 (CC_s) 转移

这样，在 $l-n$ 到 l 之间的指令就改变不了 BE 所需的条件码。然而，这要增加编译程序的负担，

除了把 DEC 指令由 $l-1$ 移到 $l-n$ 外, 还需能置定好相应的特征位。

三、加快短循环程序的处理

由于程序中广泛采用循环结构, 流水机器多采取特殊措施以加快循环程序的处理。各种措施的基本出发点, 一是如何使整个循环程序都放入指令缓冲器 (当然是程序长度小于指令缓冲器容量的短循环程序), 以减少执行循环程序时的访主存次数; 二是考虑到对循环程序的出口分支, 由于返回到循环程序本身的概率要比转到另一个程序的概率高得多, 因此对循环程序出口端的条件转移指令, 若按通常方法处理, 那如前所述会出现断流现象, 因此应尽可能使指令缓冲器或指令堆栈内的这种循环程序能首尾联结连续流动, 连续解释。

360/91 为上述第一点设置了“向后 8 条”检查。对于成功转移的条件转移指令, 若转向去址与条件转移指令地址之间相隔 8 条指令或更少, 那就认为是遇到了短循环程序, 就要把从转向去址到条件转移指令地址间的这段程序全部搬入指令缓冲器内。为上述第二点设置了“循环方式”工作状态 (使出口端的条件转移指令联向循环程序的始端) 以替代一般情况下的“条件转移方式”工作状态。采取这些措施后, 循环程序的执行时间可以缩短 2~3 倍。

2.3-3 流水机器的中断处理

中断往往是不能予知的。所以, 对流水机器, 从中断第 i 条指令, 到把中断程序由主存调入指令缓冲器, 并开始在流水线中流动, 其断流时间会是很长的。从这点看, 中断和条件转移是相似的。然而, 由于中断的出现概率要比条件转移的出现概率低得多, 因此流水机器处理中断的关键不在于如上述条件转移处理那样, 是如何缩短断流时间, 而是如何处理好断点现场及中断后的恢复问题。本小节就只分析这点。

按照中断处理的应有规定, 在处理第 i 条指令的中断时, 由于要用到的中断现场应是这第 i 条指令的现场, 所以在中断处理前, 第 $i+1$ 条本应不准执行。然而, 由于流水机器是同时解释多条指令, 如图 3.24 所示, 当第 i 条指令在解释过程中, 由于诸如地址错、访存错、传送错、运算错等原因, 而在例如第 5 段发中断申请时, 第 $i+1$ 、 $i+2$ 、 $i+3$ 、 $i+4$ 条指令都已进入流水线被解释了, 对于异步流动的流水线, 这 4 条指令中的有些指令甚至可能在第 i 条的前面流动。这样, 如何取得准确的断点现场 (指的是送给中断处理程序的应是对应于第 i 条指令的中断现场, 如第 i 条指令的程序状态字等) 及在中断处理后如何能恢复到原有现场 (指的是在中断处理完, 重新解释第 i 条指令时, 它及它之后在中断前进入过流水线的第 $i+1$ 、 $i+2$ 、 $i+3$ ……等条指令的状况应和它们未进入流水线时的一样) 就复杂了。

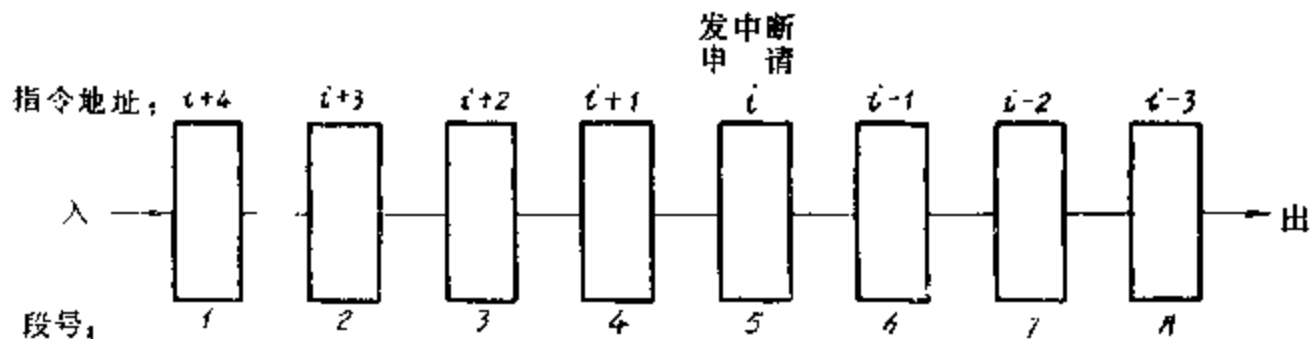


图 3.24 流水机器的断点现场问题

有些流水机器, 如早期的 IBM360/91, 为简化中断处理, 采用了所谓“不精确断点”方法: 不论第 i 条指令在流水线的哪一段发出中断申请, 那时还未进入流水线的后续指令就

不再允许其进入，但已在流水线内的所有指令(对图 3.24，包括 $i+1$ 、 $i+2$ 、 $i+3$ 、 $i+4$)都仍然流动到执行完毕，而后才转入中断处理程序。这样，虽然中断处理程序是对第 i 条指令在第 5 段发出的中断申请进行处理，但给它的断点现场却不全是对应第 i 条的，也有是第 $i+4$ 条的，即断点是不精确的。采用这种中断处理方法，只当第 i 条指令是在第 1 段发的中断申请(如非法操作码)，断点才是精确的。

这种“不精确断点”法对程序设计者当然不方便，在程序排错时更是如此。这曾是程序设计者非难流水机的一个方面。因此，后来的流水机器多采用“精确断点”法，如 Amdahl 470 V/6 就是如此。对它，不论第 i 条指令是在流水线中哪一段发的中断申请，给中断处理程序的现场都全是对应第 i 条的；而且第 i 条后已进入流水线的指令的原有现场都能恢复。

“精确断点”法需采用很多后援寄存器以保证流水线内各条指令的原有状态都能保存和恢复。但这些寄存器多是第六章要讲到的“指令复执”所需要的；因此，对要求能复执的机器，易于实现“精确断点”法。

2.4 向量的流水处理

采用向量数据表示的好处在第二章 § 1.3-2 已讲过。在这节，只是结合七十年代设计得较好的向量机器 CRAY-1，从控制角度简述向量数据表示的流水处理。关于向量并行处理技术的详细论述则已超出本书的范围。

前面已经多次讲过，只当流水线能连续流动时，才能获得高的吞吐率和效率，而对向量数据表示的流水处理是最适于发挥流水线的效率。由于对每个向量的所有元素都执行相同的操作，而且每个向量的各个元素之间又是相互无关，这样，只要能由存贮系统不断地取得成对的元素，流水线就能以每拍送出一个结果元素的最高吞吐率连续流动。

对于向量运算，例如：

$$D = A \cdot (B + C)$$

A 、 B 、 C 、 D 都是向量，长度都为 N ；那应该采用什么样的处理方式才能发挥流水线的效能呢？显然，如果采用逐个求 D 向量元素的方法，即访存取出 A_i 、 B_i 、 C_i 元素，按上述算术表达式求出 D_i ；再取 A_{i+1} 、 B_{i+1} 、 C_{i+1} ，求 D_{i+1} 的办法，即其处理顺序为：

$$D_1 = A_1 \cdot (B_1 + C_1)$$

$$D_2 = A_2 \cdot (B_2 + C_2)$$

⋮

$$D_i = A_i \cdot (B_i + C_i)$$

⋮

$$D_N = A_N \cdot (B_N + C_N)$$

那是不可能使流水线能连续流动的。因为上述表达式至少需用二条指令：

$$B_i + C_i \rightarrow K_i$$

$$K_i \cdot A_i \rightarrow D_i$$

实现。这就要出现数相关，即出现如本章 § 2.2-3 求 Z 实例中，需把流水线输出回投到输入，使吞吐率下降的情况；如果再加上要执行不同的运算(加和乘)，需进行如图 3.18 的功能切换，那流水线的吞吐率还会进一步恶化。

因此，只有采用能对每个向量的所有元素执行相同操作，即在对整个向量按相同操作都

执行完之后，再转去执行别的操作，才能发挥流水处理的效能。亦即应改为按：

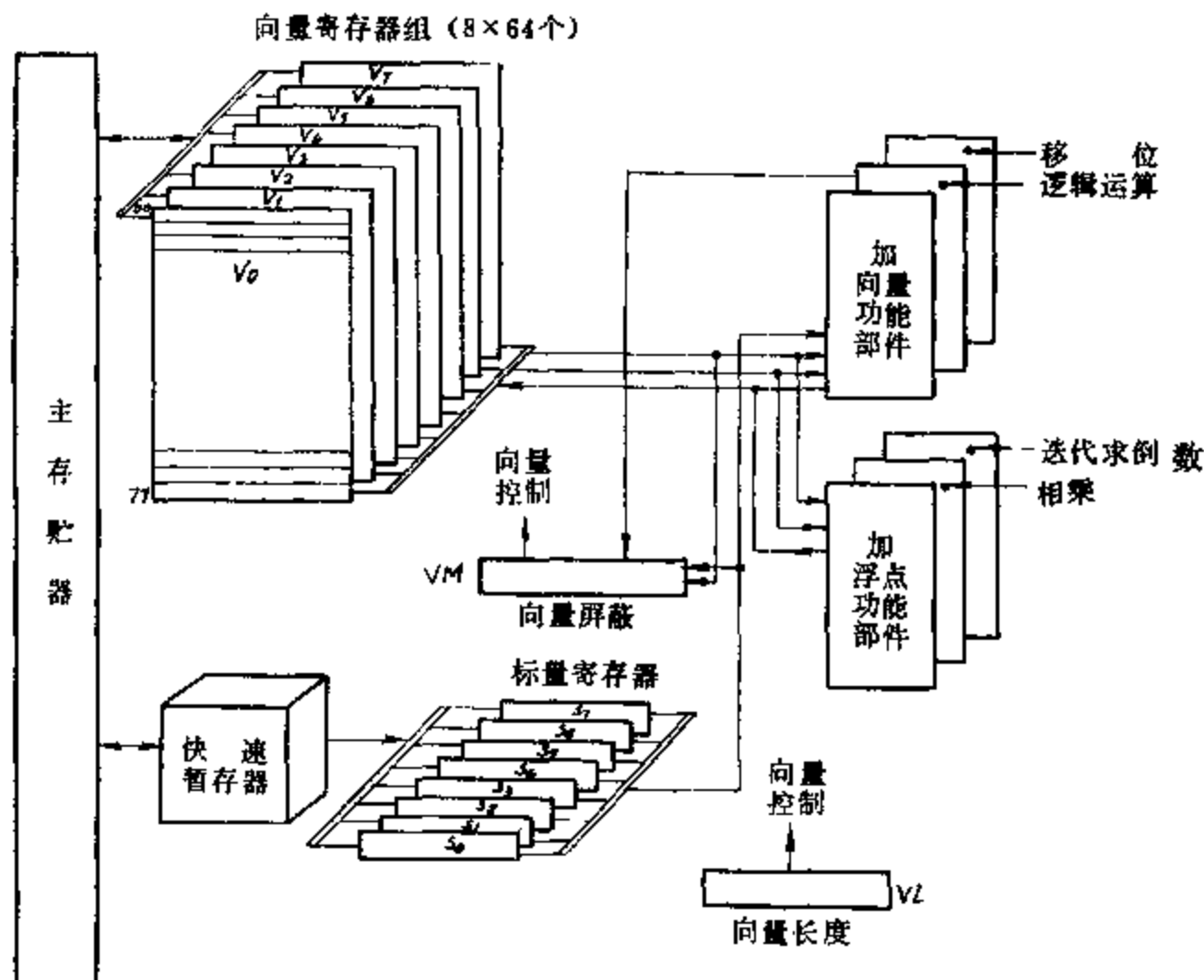
$$B + C \rightarrow K \quad (1 \text{ 到 } N)$$

$$K \cdot A \rightarrow D \quad (1 \text{ 到 } N)$$

求值。这样，在算 $(B+C)$ 时，因为是对 B 、 C 向量的所有元素都执行相同的加法操作，而且 B 、 C 向量的各个元素之间又是相互无关，所以，不会出现要求把流水线输出回授到输入的情况，使流水线可以连续流动，从而使其效能得以发挥。在算 $K \cdot A$ 时也是如此。这里，功能切换总共只需进行一次，而不是前面那种每求一个 D_i 都需进行一次切换。上述这种处理方式，称为向量的流水处理，向量机就采用这种处理方式。这样，从流水线输出端可以每拍取得一个结果元素。

然而，这只有在流水线输入端能不断地（每拍）取得成对的元素的情况下才能每拍取得一个结果元素。由于 N 值不会过小，因此早期的向量机，向量运算指令的源向量和目的向量都是在主存内；这就要求显著提高主存与流水处理机之间的信息流量才能支持流水线的连续流动。如七十年代初问世的 STAR—100 就不得不把磁心主存设计成是 32 个体交叉，且每个体的数据宽度为 8 个字（字长 64 位），使得最大信息流量达每秒二亿字（关于多体交叉详见第五章）。

然而，主存并非单是为中央处理机所使用，很多通道也要用，要保证源元素的连续供应和结果元素的按时存入主存，这很不容易；而且，多体交叉的实际流量往往要比最大流量小得多。因此，把主存直接连到流水线输入、输出端的做法不是好的方案，有可能支持不了流水线的连续流动。这样，随着半导体存储器片子价格的持续下降，在 STAR—100 之后，于七十年代中期问世的 CRAY—1 向量机就改为把流水线输入、输出端连到容量足够大的向量



寄存器组，且向量寄存器组与主存之间是成组传送的。

CRAY—1 机器的向量流水处理部件简图见图 3.25。可为向量运算使用的功能部件为：整数加、逻辑运算、移位、浮点加、浮点乘、浮点迭代求倒数；它们都是流水处理部件，且六个部件可并行工作。向量寄存器组的容量为 512 个字，分成 8 块。每个 V_i 块可存元素数达 64 的一个向量。

为了能充分发挥向量寄存器组和可并行工作的六个功能部件的作用以及加快向量处理，CRAY—1 设计成每个 V_i 块都有单独总线可联到六个功能部件，而每个功能部件也各有把运算结果送回向量寄存器组的输出总线。这样，只要不出现 V_i 冲突和功能部件冲突，各个 V_i 之间和各个功能部件之间都能并行工作，大大加快了向量指令的处理，这是 CRAY—1 向量处理的显著特点。

所谓 V_i 冲突指的是并行工作的各向量指令的源向量或结果向量的 V_i 有相同的。除了相关情况之外，就是出现源向量冲突，例如

$$V_4 = V_1 + V_2$$

$$V_5 = V_1 \wedge V_3$$

这二条向量指令也不能同时执行，需在第一条向量指令执行完，释放 V_1 后，第二条才能执行。这是因为这二条指令的源向量之一虽然都是取自 V_1 ，然而二者的首元素下标可能不同，向量长度也可能不同，难于由 V_1 同时提供二条指令所需的源向量。

所谓功能部件冲突指的是同一个功能部件被一条以上的并行工作向量指令所使用。例如，

$$V_4 = V_2 * V_3$$

$$V_5 = V_1 * V_6$$

这二条向量指令都需用到浮点相乘部件，那就需在第一条指令执行完毕，功能部件释放后，第二条才能执行。

CRAY—1 有四种向量指令如图 3.26 所示。第 I 种，每拍从 V_j 、 V_k 块顺序取得一对元素送入需 n 拍完成的功能部件，各种功能部件的 n 值不同，其输出也是每拍送进 V_i 块一个

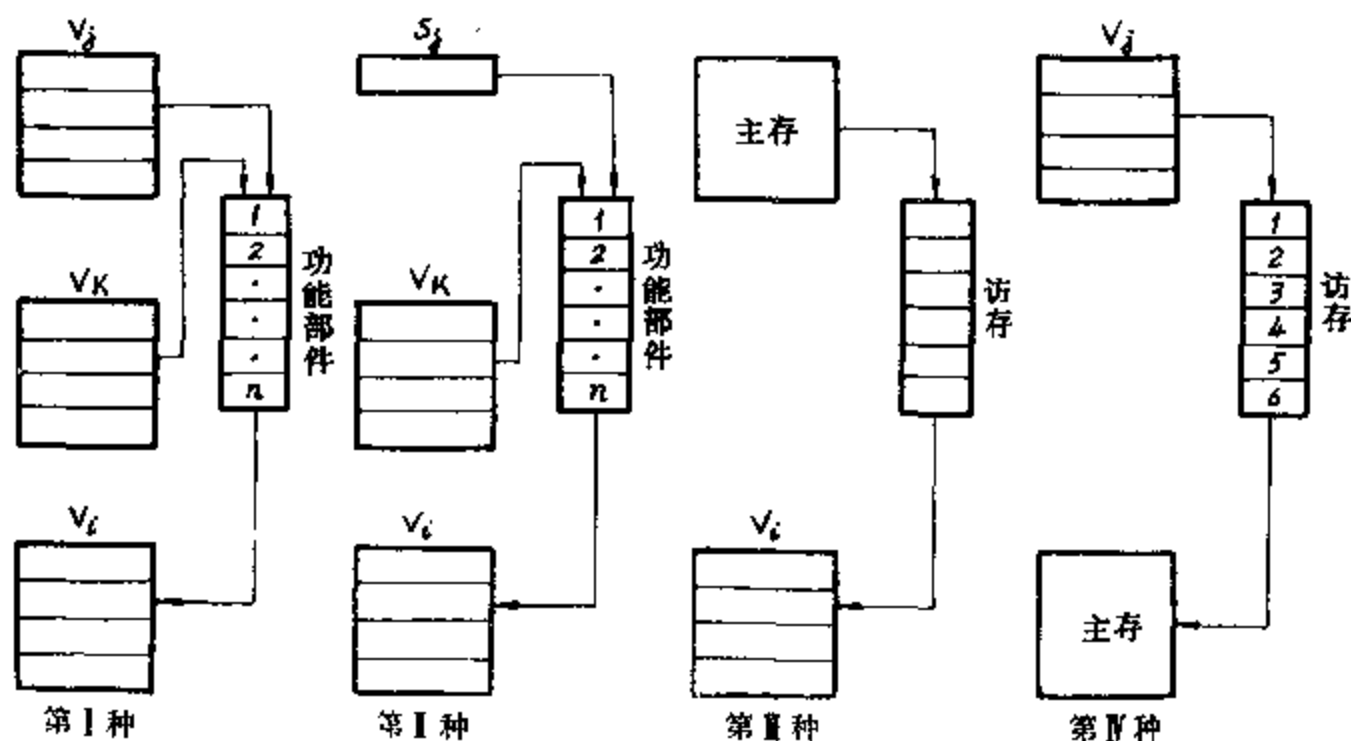


图 3.26 CRAY—1 的四种向量指令

结果元素。元素对的个数由 VL (向量长度) 寄存器指明。向量屏蔽寄存器 (VM) 为 64 位, 每位对应 V 的一个元素; 在向量合并或测试时, 由 VM 控制对那些元素进行合并和测试。一条指令至多只能处理 64 对元素 (对应每块的容量), 若向量的长度大于 64, 则向量寄存器就存不下, 需用向量循环程序分段由主存调入并分段处理。第 II 种和第 I 种的差别只在于它的一个操作数取自标量寄存器 S_j 。大多数向量指令都属这二种, 由于它们不是由主存, 而是由向量寄存器取得操作数, 所以流水速度可很高, CRAY-1 的每拍为 12.5^{ns} 。第 III、IV 种是控制主存与 V 向量块之间的数据传送, 存、取一个字 (元素) 需化 6 拍。

CRAY-1 向量处理的另一个显著特点是通过链结结构可使相关的向量指令也能并行处理, 只要不出现功能部件冲突和源向量冲突。例如, 对前述向量运算:

$$D = A * (B + C)$$

若 $N \leq 64$, 向量为浮点数, 则在 B、C 取到 V0、V1 后, 就可用以下三条向量指令求解:

1. $V3 \leftarrow \text{存储器 (访存取 A)}$
2. $V2 \leftarrow V0 + V1$ (浮点加)
3. $V4 \leftarrow V2 * V3$ (浮点乘, V4 存 D)

第 1, 2 条指令无任何冲突, 可以并行执行; 但第 3 条指令与第 1, 2 条指令有数相关, 本不能并行执行, 但若能把第 1, 2 条指令的结果元素直接链结进第 3 条指令所用的功能部件, 那第 3 条指令就能与第 1, 2 条指令并行执行。它们的链结过程如图 3.27 所示。

CRAY-1 是把元素送往功能部件及把结果存入 V_i 都需一拍。由于第 1, 2 条指令之间没有任何冲突, 可以同时执行, 而“访存”拍数正巧与“浮加”的一样; 因此, 从访存开始, 直至把第一个结果元素存入 V4, 所需拍数 (亦称为链结流水线的流水时间) 为:

$$1 (\text{送}) + 6 (\text{访存}) + 1 (\text{入}) + 1 (\text{送}) + 7 (\text{浮乘}) + 1 (\text{入}) = 17 \text{ 拍}$$

此后, 就是每拍取得一个结果元素存入 V4。显然, 这要比第 1, 2 条指令全执行完, 所有元素全进入 V2, V3 后, 才开始执行第 3 条指令要快得多。

通过这种链结技术使得 CRAY-1 流水线能灵活组织, 从而更能发挥流水技术的效能。

CRAY-1 的向量指令还可做到“源 V”和“结果 V”是同一个, 这种向量递归操作和前述的链接特性对于实现诸如求向量点积等是很有好处的。

上面我们结合 CRAY-1 介绍了向量流水机的结构特点。然而, 要使程序能充分发挥硬件所提供的这些特点却是很不容易的, 它必然要对语言结构和软件设计提出新的要求。例如, 它希望高级语言能增设向量运算符 (如向量加、向量乘等); 不然, 程序设计者在编制高级语言程序时, 要把向量运算通过 DO LOOP 实现, 而编译程序反过来却又要将 DO LOOP 型语句变换成向

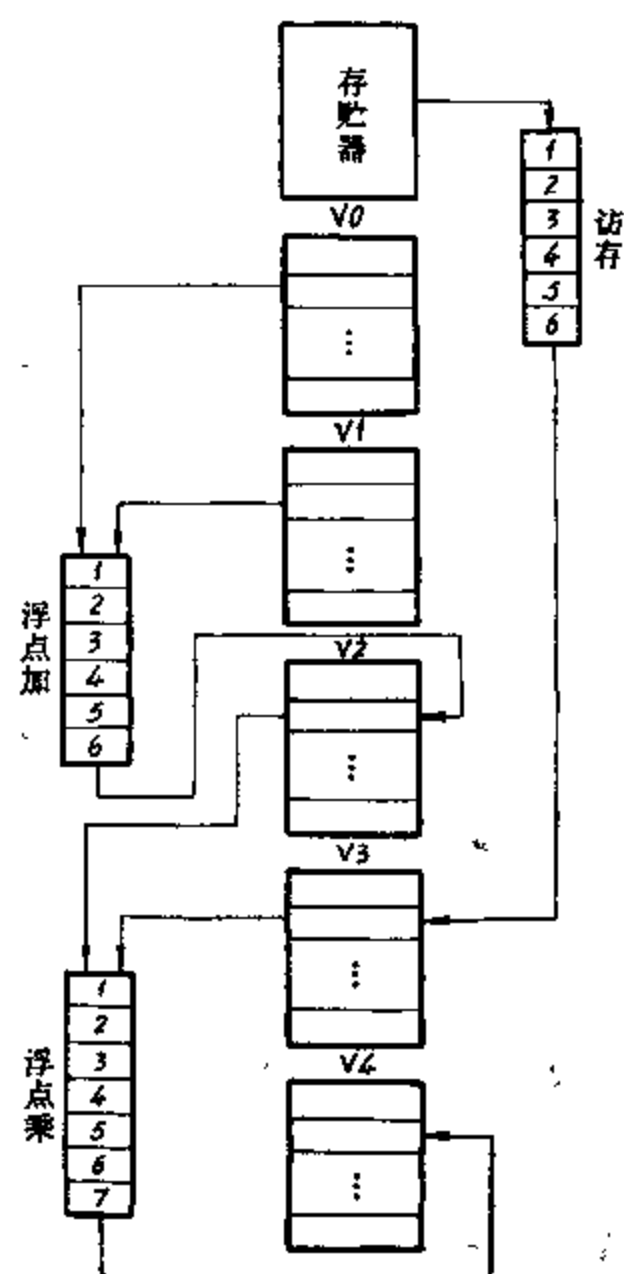


图 3.27 通过链结技术实现并行执行

量型的机器语言去执行。例如，

$$C = A + B$$

的向量运算，程序设计者是用

```
DO 20 I=1, N
20 C(I)=A(I)+B(I)
```

实现，但编译程序却又要把它编译成

```
VECT-BEGIN
A,B,C=VECTOR(1...N)
C=A+B
VECT-END
```

去执行。

另外，优化的目的程序必然要和向量流水机器的具体结构特点密切相关，这要使编译程序的设计复杂化。例如，对图 3.15 所示的 ASC 结构，那就要求把目的程序应优化成相加和相乘指令分别合并，以减少流水线功能切换所费时间；然而，CRAY-1 却无此要求，因为它有独立的能并行执行的相乘、相加功能部件。但是，它却要求编译程序能充分发挥多功能部件和上述链结能力所能提供的多条向量指令可并行执行的特点。

§ 3 分布处理方式简述

先讲讲本节所指的“分布处理”的含义和概念，接着结合智能终端的简单介绍讲述有关的概念，最后简述局部式分布处理系统结构的构成原理和基本概念。

3.1 引言

大规模和超大规模集成电路的发展，尤其是微处理器的发展，不只是促进了计算机在更广阔领域的应用；而且，还由于可以很便宜地在单片上、一个组件内获得原先是一个处理机的能力，因此必然地影响、并将更深刻地影响到计算机本身的系统结构。现在，从硬件上看，在一台计算机内或是在一个计算机系统内设置多个处理机是完全可能和合理的。从构成巨型机出发，主要着眼于提高机器运算速度的多处理机和多机系统将在第七章讲述。我们在这里所讲的分布处理计算机系统虽然也具有多台处理机，但主要是着眼于提高计算机系统的性能价格比。

什么样的计算机系统是分布处理的，这有各种各样的说法。这是因为分布的含义可以包括硬件的分布（分散）、数据（数据库）的分布、处理过程或能力的分布以及控制（包括操作系统）的分布。有人认为，分布处理系统指的是由几个可编程序处理机组成的计算机系统，在这个系统中，通常的每个计算任务要用到二个以上的处理机。这种定义当然是笼统的。

从广义上看，从要求来看，分布处理系统内的各个处理机，其任务应是可以动态地分配的，即每个处理机的功能及其所执行的任务不应是固定不变的；此外，如何使各个分布点上的数据能为各个处理机共享，能为整个计算机系统使用，应是设计分布处理系统的主要目标之一。然而，要实现这些，还有许多理论和实践上的难题，在许多方面的研究还仅仅是开始。关于这种分布处理系统的论述不属本书的原定范围。

我们在这里所讲述的分布系统，其含义是较窄的，或是说，还是比较低级的。它利用了大规模、超大规模集成电路技术和微处理器的成果，努力实现硬件的分布、处理的分布以及控制功能的分布，它符合上述的笼统定义。然而，它并不是着眼于构成追求高速和超高速的巨型机，也还不是把实现动态功能分配和数据共享作为主要的设计目标。它是立足于基本上不改变从用户所看到的系统结构，要求现有高级语言仍然能够使用，努力使得用高级语言编写的已有应用程序能够在该系统中运行；它主要着眼于如何通过分布处理，努力提高计算机系统的性能价格比。在这种分布处理计算机系统内各个处理机的功能是专用的，固定不变的；这种分布处理系统要求各个分布点尽可能地利用其自己的处理机实现就地处理，以减少各分布点之间、各处理机之间的信息通讯。这种分布处理系统本身是短程网络，当然，它也可以与现有计算机系统一样接到计算机网上。还有，在这种系统内一个进程的分布实现对用户来讲是透明的，用户只需指出要做什么，而不必指明应由谁来完成。总之，它基本上仍然是对现有计算机系统的改进，而不是突变性的变革。我们把这种分布处理系统称之为局部式分布处理系统。

局部式分布处理系统比之于目前集中式的一般计算机系统，其主要优点在于它利用集成度不断提高、硬件价格持续下降的特点，构成高效率的各种专用功能处理机以实现负荷分担，使得在同样的器件速度，同样的工艺、装配条件下，计算机系统的速度得以提高。还有，它具有更好的灵活性，更易于实现以前讲过的把各种专用机的成果以专用功能部件的形式加到通用机系统上；显然，它还有利于模块化的实现。而上述这些又是在尽可能保持应用软件兼容的前提下实现的，从而易于为已拥有大量应用软件的用户所接受，便于获得推广。

实现这种分布处理系统的难点在于“联络（通讯）”和“控制（调度）”。既然一个进程是要分布于多个处理机内实现，那就要解决好各个处理机之间应该如何联系和通讯的，应该如何尽可能降低对各分布点间联接网络频宽的要求。既然各个专用功能处理机能够并行执行，那就要解决好采用什么样的算法，如何控制才能充分发挥这种并行性。还有，虽然各个分布点应有就地处理的能力，然而，它的能力毕竟不应过强，以免造价过高、利用率过低；因此还需要解决如何实现任务分配，如何在就地处理不了时，能转到别的处理机去处理等等。为了实现这些，除了各个处理机都需有各自的操作系统之外，还需设计好控制整个分布处理系统的总操作系统（需由单独的操作系统专用机来实现）。这样，实现这种分布处理系统的关键之一在于软件（包括微程序软件）设计。需要研究从基本片子到整个系统的设计工具。

应该说，这种分布处理系统的思路并不新。在第一章讲过，I/O 处理机的出现就是这种思路的萌芽。从操作系统看，大家知道，目前集中式计算机的操作系统是集中地由中央处理机实现。由于操作系统的功能愈来愈强，致使操作系统十分庞大。既然操作系统的主要任务是管理好软、硬件资源，若软、硬件资源并不是集中在一个地方，那当然会想到为什么不能把部分操作系统分散靠近到各有关资源上呢？以便提高操作系统的执行效率、减轻中央处理机的负担和缩短辅助操作的时间。其实，六十年代中期问世的 CDC-6600 就采用了这种分布实现的思路，它的 I/O 处理机（总共可达 10 台）除了控制 I/O 操作之外，还执行操作系统的某些功能。七十年代 CDC 进一步发展了这种思路。当然，I/O 处理机能力的加强，是可能会出现利用率不高的情况；因此，当处理机价格在整个计算机系统价格中还占据较高比例时，上述分布处理系统的思路是难于被计算机厂家和用户所接受的。但是，现在情况不一样

了,对于价格很便宜的微处理器硬件,其利用率高低已经是次要的了,只要分布处理能使计算机系统的性能价格比得到提高,即使用了几十个微处理器,那怕这些微处理器的利用率较低也是适当的、合理的。

我们之所以把这种分布处理方式放在这一章来讲述,是因为它属于程序解释方式的范畴。前述“重迭”、“流水”技术是实现这种分布处理的重要技术。

下面我们就以把微处理器引入显示终端,构成所谓智能终端,来进一步阐述上述分布处理方式的一些思路。

3.2 智能终端简介

智能终端是从显示终端发展而来的。大家知道,从功能上看,带键盘的显示终端最初只能用键盘进行输入操作,输入或输出内容显示在屏幕上;接着,增加了编辑的功能,先是只能完成字符删除,很快也能进行字符插入。上述这些功能的设置只是从输入的需要出发,终端不需要有什么处理能力。进一步,要求终端还能指明词法上的某些错误,如用闪烁、增加亮度等来指明词法出错字符;还要求终端具有数据格式变换和码制变换的能力,即终端送往主机的是变换过(或压缩了)的信息。这些就要求终端具有原属中央处理机(或是I/O处理机、前端机)的某种处理能力;就是说,要求把这种处理能力由中央处理机分布到各个终端去。这样,终端内的存贮器容量大小就不只是根据屏幕显示的需要,还要满足这种处理的需要。更进一步,某些用途还要求终端能不依赖于主机把原输入了的信息重调出来以及不是实时地把一页一页输入信息送往主机,而是经过一段时间,积累了一批数据后再一次发送。这些就要求终端内还得有如软盘或盒式磁带这种磁性存贮器。有的用途还要求给终端配上打印机,它既可打印出存在终端软盘或盒式磁带上的内容,也可打印出由主机送来的信息,这就要求终端能有与打印机相联的接口及相应的控制环节。为了能配上磁性存贮器和打印机,当然要求进一步增强终端的处理能力。

上述种种要求,虽然可以用分离组件构成的数字环节实现;然而,采用微处理器实现,通过对它的分时使用,既可节省硬件造价,又能提高灵活性。这样,就逐步地将微处理器引入显示终端,并给终端配上交互式操作系统以控制上述功能。开始时,只给微处理器配上容量很小的ROM、RAM,终端只具有固定的处理能力;而后,随着对这种终端的使用经验的积累,希望进一步增强它的功能和灵活性,这就要求扩大RAM的容量,而RAM片子价格的持续下降也支持了这种努力。这样,包括操作系统的终端软件就不只是存在ROM内,还可存在软盘或盒式磁带内,并按需要调入RAM;这样就可给终端配上编译软件,使得一般高级语言程序可在终端就地编译,而一些不复杂的题目可不必调用主机,在终端上即可直接求解。

至此,终端具有用户程序可编能力,也主要因为这一点,人们普遍同意把这种终端称为智能终端。有了用户程序可编能力,终端就能按照用户的需要扩展其功能。例如,强功能的用户定义键就是其中很有用的一种,它使得用户按下一个键符就能按照用户需要执行复杂的输入和控制操作。此外,磁性存贮器就不只是暂存用户输入的数据,还可用于构成局部的小数据库,使得不必调用主机,由终端就可对它的局部文件进行检索、查找。图3.28是DEC公司1979年问世的PDT-11智能终端系列的PDT-11/130原理框图。

图上这些,包括盒式磁带机(容量512K字节)在内,都装在一个带显示的终端内。1K

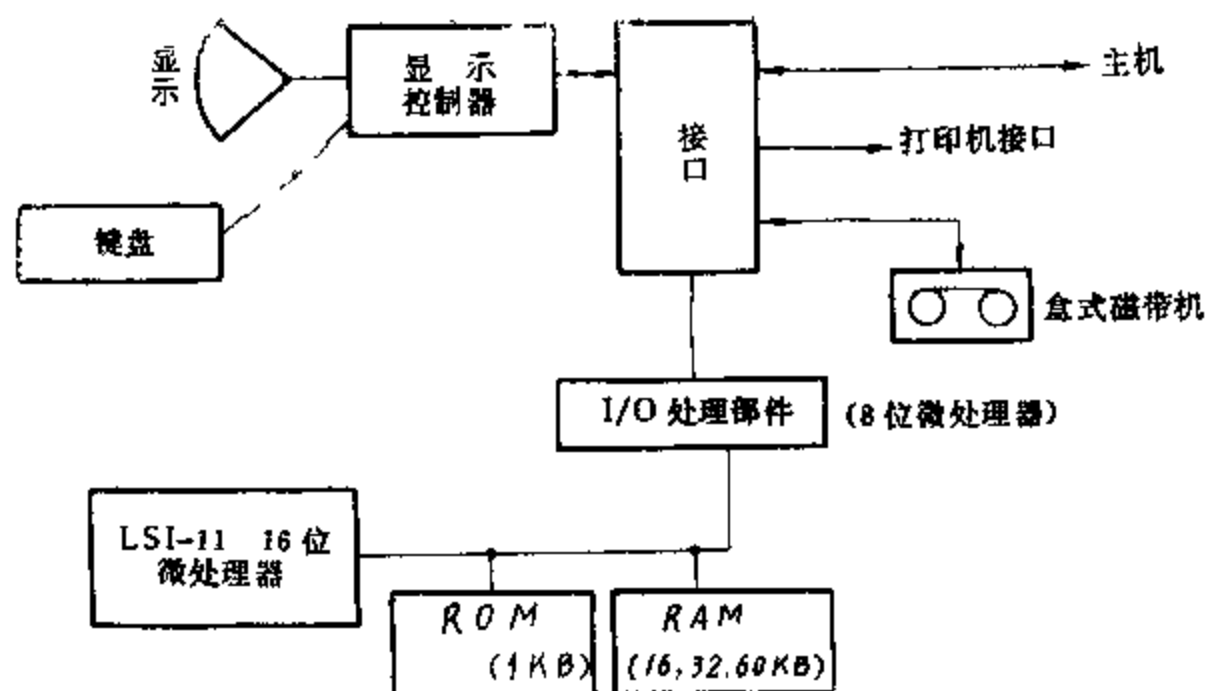


图 3.28 PDT-11/130 智能终端原理框图

字节的 ROM 采用现场型 PROM 片子，它存了诊断、引导、加载和启动等程序，还留一部分给终端使用者用；RAM 的容量最大可达 60K 字节；接口部件能与主机同/异步接口，直接控制磁带机以及与打印机接口相连；构成 I/O 处理部件的 8 位微处理器，在第一章 § 4.1-2 已提过，是作为现场型片子用，用于控制与接口及控制器的联系；终端的运算、处理能力是由 LSI-11 微处理器实现。由于 LSI-11 是 PDP-11 的微型化，因此智能终端与主机 PDP-11 兼容，有些原在主机 PDP-11 运行的汇编语言程序可以移到智能终端执行。

PDT-11/130 智能终端本身配有小型的交互式操作系统，可接受 FORTRAN、BASIC 和宏汇编等语言。PDT-11/130 主要用于源数据的收集及某些予处理，如检查信息的正确性、进行编辑等等，而后把予处理过的正确的、或压缩了的信息传送到主机，帮助主机解脱某些费时的负担，从而提高整个计算机系统的处理能力。有些比较简单的应用程序还可直接在智能终端算出。小的数据库（容量为 512K 字节之内）可直接存在终端的磁带机，构成局部数据库。

已公布的 PDT-11 智能终端系列有三挡，PDT-11/130 为中挡，高档的配有软盘以替代盒式磁带机。

由以上发展简述可以看出，智能终端的“智能”并不是一般理解的能从经验中学习去处理新情况、新问题；而是相对于原先一般“哑”终端而言的“智能”。“智能”只是指的这种终端不只具有一般“哑”终端的输入和显示能力，还具有一定的存贮、分析和处理能力，主要特点是终端本身具有可编程序能力。显然，智能终端和“人工智能”并无联系；不论人们对分布处理如何下不同的定义，它应是属于分布处理的范畴。它把原先由中央处理机集中进行的原始数据予处理分布到各个终端去完成，这有助于缓和 I/O 瓶颈；由于终端能以各种规范与主机或通讯网络联系，这也使得系统具有灵活的用户接口。由于智能终端能在没有主机支持下，实现对终端所存数据的检索，而且这些数据不仅是经终端输入的，还可以是由主机调来的，这在一定程度上可以说是数据库分布的初步。

给终端赋予局部就地处理能力是有意义的，因为数据的使用有某种局部性，即在一段时间内所用数据区变化不大。这样，若按符合局部性的一定算法，将数据由主机调到（或经输入到）终端，那终端是可能在其处理能力范围内较长时间地对这些数据独立进行处理的。这

种分布处理能力对于减轻主机负担，提高整个系统的性能价格比是有好处的。这种局部性概念其实也是构成分布处理系统的基础之一。当然，应该做到在智能终端处理不了时，能自动地把任务“上交”给主机。要实现这点，关键在于设计好终端操作系统和主机操作系统的接口与联系。其实，智能终端性能的高、低，关键就在于所配的软件。

我们在前面讲述智能终端的形成过程时，是从如何使终端的“智能”向原先中央处理机的处理能力接近来讲的。然而，这主要是从分布处理的观点和需要来讲的，这也是我们为什么把智能终端放在分布处理方式这节来讲的原因。但是，就智能终端本身来讲，绝不是说它的智能愈接近于中央处理机的能力就愈好。智能终端应具有什么样的智能，归根结底取决于它的应用领域和需要。智能终端毕竟是“终端”，它一般是不会作为独立的计算机使用。因此，它应是在终端价格的基础上，尽力在只增加少许造价的情况下增加其“智能”，使它的性能价格比能比一般微型机强；它亦应该系列化、模块化。

至今，智能终端的销售额仍然不高，就是设计得较好的 PDT-11 也是如此。因此，对智能终端的设计应慎重，要根据用户和市场的需要来设计，而不是片面追求高“智能”。总之，从分布处理的观点来看，智能终端的出现是一个有意义的进展。随着它的应用的扩大和经验的积累，智能终端必将在发展分布处理系统中起重要作用。

3.3 局部式分布处理系统结构简介

我们在 § 3.1 讲了局部式分布处理系统的基本概念，图 3.29 是它可能的一种结构原理图。

图中的每个处理机都有其自己的局部存贮器，主存模块用于存贮公共数据，也可用于各处理机间的信息变换。

前端处理机与近、远程终端和智能终端相联，用于处理交互式用户，各智能终端之间是分布处理；前端处理机还用于控制远程外设并处理与计算机网络的联结。高级语言源程序经

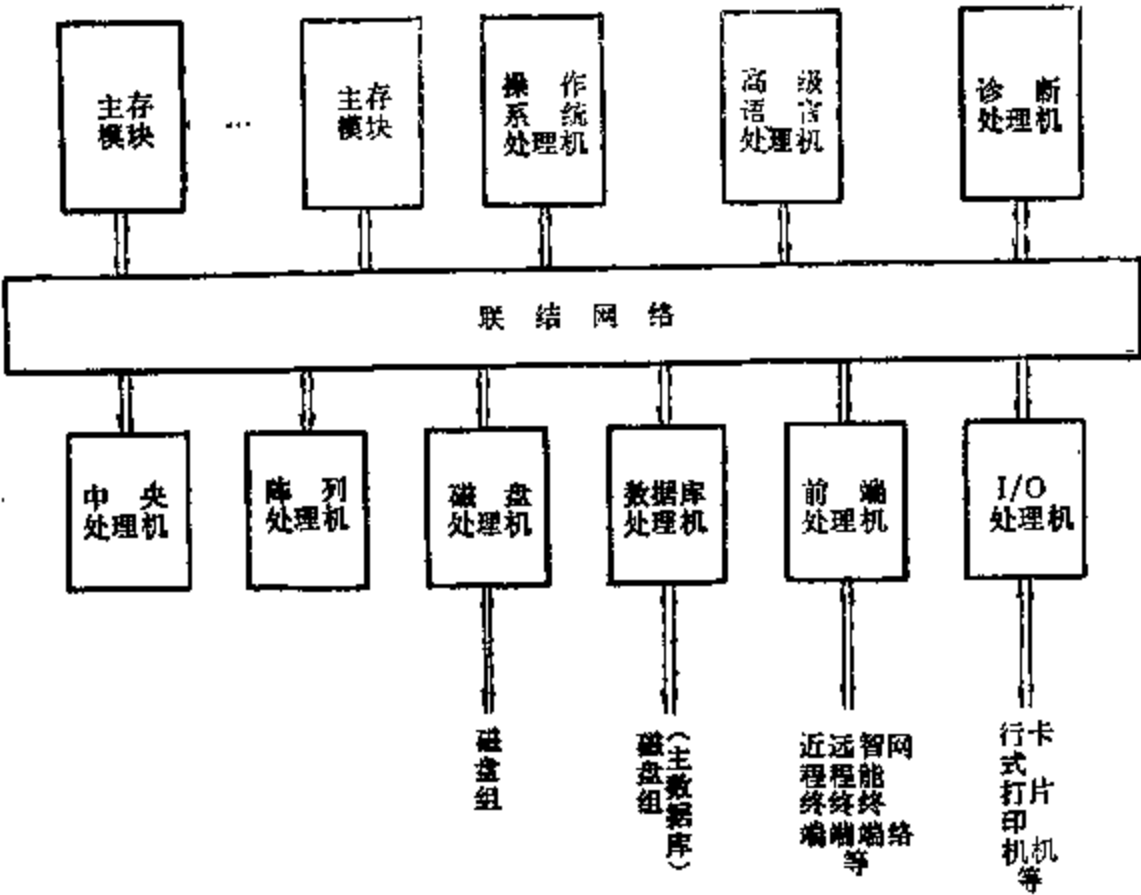


图 3.29 局部式分布处理系统结构

前端处理机送往高级语言处理机(可参看第二章 § 3.2 节), 翻译成目的程序后送往中央处理机进行运算。如果高级语言源程序已在智能终端翻译好了, 但智能终端又处理不了, 则可经前端处理机送往中央处理机处理。各个智能终端上的局部数据库可经前端处理机、数据库处理机与主数据库相联。

I/O 处理机与行式打印机、卡片输入机等控制器相联, 用于处理经这类外设的输入与输出。高级语言源程序经 I/O 处理机送往高级语言处理机, 而运算结果则由中央处理机或是送往前端处理机经终端或远程外设输出, 或是送往 I/O 处理机由打印机输出。

数据库处理机用于管理主数据库, 对主数据库的查找和装入都是在这个处理机上软、硬结合地实现。存主数据库的磁盘机是由装有微处理器的“智能”控制器控制, 以实现在磁盘机上的就地相联查找。

磁盘处理机用于处理外存贮器与主存贮器之间的信息传送, 它能实现把外存逻辑地址变换成外存物理地址; 根据需要, 经磁盘处理机还可将外存贮器内容直接送往各个处理机的局部存贮器。

中央处理机完成解题运算, 它除了有局部存贮器之外, 还有高速缓冲器。阵列处理机是专设计用于高效地处理阵列数据的, 可把它看成是中央处理机的扩展部件, 当中央处理机遇到阵列运算时, 就转到阵列处理机高效执行。这种阵列专用处理机在七十年代末期就已出现, IBM 系统和 PDP-11 系统都有。阵列处理机不一定要如图所示, 经联结网络接往中央处理机, 也可以直接与中央处理机相连以减轻联结网络的负担。

由上述关于各个处理机基本功能的简述中可以看出, 各个处理机(甚至包括阵列处理机)本身都需有各自的局部操作系统; 就是说, 这种分布处理系统的操作系统是分布实现的。然而, 各个处理机之间又是密切相关的, 一个任务是要经过多个处理机才能实现。因此, 必须有一个管理各个处理机协调工作的主操作系统。操作系统处理机就是用于软、硬结合地实现这个主操作系统, 它能够处理各种作业控制语句。关于操作系统处理机的一些思路, 我们在第二章 § 3.2 提到过。

诊断处理机用于对整个分布处理系统以及各个处理机的对外接口、基本功能进行联机或脱机诊断。

联结网络的性能如何是实现这种分布处理系统的主要关键之一, 这个网络不只要传送数据信息, 还要传送控制信息, 它必然是整个系统的瓶颈。为此, 一方面要尽力提高联结网络的通讯频宽(显然, 采用一般的总线结构是肯定适应不了的); 另一方面, 又要尽可能减少各处理机之间的信息来往, 要尽量发挥各个处理机的就地处理能力, 做到信息在各处理机之间是成块传送, 而不是某个处理机在进入处理某个任务后, 还需经常地与其它处理机交换信息。给各个处理机配上各自的局部存贮器和局部操作系统有助于实现这点。

可以看出, 这种分布处理系统是由已有计算机系统进一步分布, 演变而成的。例如, 我们在第一章 § 2.2-2 讲过的 I/O 处理机在这里就进一步分布成磁盘处理机、前端处理机和 I/O 处理机, 并且加强了这些处理机的软件功能。又如, 原先全由中央处理机完成的任务, 在这里分布到多个处理机(阵列处理机、高级语言处理机、操作系统处理机、数据库处理机和中央处理机等)去完成。

这样, 在这种分布处理系统中, 有各种各样的处理机, 而有的处理机又不一定只有一台(如磁盘处理机、前端处理机、I/O 处理机以至中央处理机等)。如果各个处理机的动作都

需由用户指明,那这种分布处理系统是不会被用户接受的。因此,这种系统的对外接口很重要,一定要做到使系统的“分布处理”特征对用户程序是透明的。显然,透明性要求会使系统的控制功能复杂化。如果再加上一个进程需用到多个处理机,各个处理机可以并行执行,而在各个处理机之间又采用“流水”技术(参看本章图 3.14),那光是实现进程同步这一点就很不容易。更何况对分布处理系统,要主操作系统能及时了解所有处理机的工作状态,并迅速作出反应是很不容易的。所以,在设计这种分布处理系统结构时,必须努力使硬件对操作系统的实现提供尽可能多的支持。总之,对这种分布处理系统,“控制”与“联结”的设计是关键,各个处理机既要能够“自治”,又需要服从统一的管理。

除透明性外,兼容性也应是设计这种分布处理系统的基本要求。即使这种分布处理计算机系统的性能价格比能高于已有的计算机系统,如果不具有高级语言应用程序的兼容性,那还是难于受到用户欢迎的。

我们在本章 § 3.1 讲过,如果分布处理系统各个处理机是通用的,那是更符合分布处理的含义;然而,这可能会使“控制”和“任务调度”比各个处理机是专用的更复杂。但是,处理机的专用绝不意味着各个处理机的硬件都需相互不同,都需面向其各自不同的用途来设计;如果是这样,那这种分布处理系统的硬件设计费用必然要高昂到不能被接受的地步。研究这种分布处理系统的意义就在于它的各个处理机有可能由大批量生产的现场型 VLSI 片子构成,如果能由位片型或单片型微处理器构成那是最好的,那怕是在一个处理机内就用了多个微处理器也是合适的。只有这样,这种分布处理计算机系统的性能价格比才能优于已有计算机系统。

我们讲述这节的目的在于主要从控制方式角度说明 LSI、VLSI、微处理器的出现必然会使计算机系统结构发生很大的变化;当然,这种局部式分布处理系统只是可能的一种。

主要参考文献

- [1] DJS-240 系列机设计资料, 1975.
- [2] Tien Chi Chen, "Overlap and Pipeline Processing," Ch. 9 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [3] J. P. Hayes, "Computer Architecture and Organization," McGraw-Hill, 1978.
- [4] D. J. Kuck, "The Structure of Computers and Computations," Vol. 1, John Wiley & Sons, 1978.
- [5] 金兰等, "并行处理计算机结构", 国防工业出版社, 1982.
- [6] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No.1, March 1977, pp. 61-102.
- [7] D. W. Anderson, et al, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," IBM Journ. of R. and D., Vol. 11, No.1, 1967, pp.8-24.
- [8] T. G. Hallin and M. J. Flynn, "Pipelining of Arithmetic Functions," IEEE Trans. on Computers, Vol. C-21, Aug. 1972, pp. 880-886.
- [9] Amdahl 470 v/6 Machine Reference Manual, MRM 100-1, 1976.
- [10] R. L. Sites, "An Analysis of the Cray-1 Computer," The 5th Annual Symposium on Computer Architecture, 1978, pp. 101-106.
- [11] R. M. Russell, "The Cray-1 Computer System," Comm. of the ACM, Vol. 21,

- No.1, Jan. 1978, pp. 63-72.
- [12] P. M. Johnson, "Cray-1 Computer System-An Introduction to Vector Processing," Cray Research, Inc., 1976.
 - [13] P. H. Enslow, Jr., "What is a Distributed Data Processing System?" Computer, Vol. 11, No.1, Jan. 1978, pp. 13-21.
 - [14] M. L. Stiefel, "What is an Intelligent Terminal?" Mini-Micro System, March 1977, pp. 50-60.
 - [15] A. Recoque, "Survey of Main Trends in Computer Hardware Architecture," IFIP 80, Oct. 6-9, 1980, pp. 115-125.
 - [16] W. G. Fosocha and E. S. Lee, "Performance Enhancement of SISD Processors," The 6 th Annual Symposium on Computer Architecture, 1979, pp. 216-231.
 - [17] J. E. Juliussen and W. J. Watson, "Problems of the 80's, Computer System Organization," The Oregon Report, Computing in the 1980's, 1978, pp. 14-23.
 - [18] H. S. Stone, "Challenging Problems in Distributed Computation," The Oregon Report, Computing in the 1980's, 1978, pp. 24-28.

第四章 输入输出系统

输入输出系统应包括输入输出设备、输入输出设备控制器以及与输入输出操作有关的软、硬件。本章是从系统结构的观点来分析 I/O 系统。

通常，要求把 I/O 系统的系统结构设计成对应用程序员是透明的。要认识到，从程序的输入到运算结果的输出，其整个过程中，耗费于 I/O 系统的时间往往是最多的；然而，在相当长的一段时期内，系统结构设计者对 I/O 系统是重视不够的，在确定 I/O 系统的软、硬件功能分配时，往往过多地把任务分给系统软件去完成，不仅加重了 CPU 的负担，而且还难以减少耗费于 I/O 系统的时间比例。

最初的计算机是接单用户设计的，I/O 系统的设计任务主要是要解决好运算器、主存和 I/O 设备在速度上的巨大差距。那时，输入、输出过程是由程序员自己安排的。而后，随着分时系统的出现，由于是多达几十个甚至上百个用户要分享宝贵的主存和 CPU 资源，那每个用户程序的输入和其运算结果的输出，当然不能由各用户自己安排，而是由 I/O 系统来处理。这样，I/O 系统的性能好坏不仅要影响到各个用户从程序输入到运算结果输出的时间，而且还会影响到 CPU 和主存的利用率高低。

随着计算机系统的不断发展和应用领域的进一步扩大，系统所要求的输入输出数据量迅速增大、数据传送的速度显著提高、输入输出设备的种类日益繁多，从而导致输入输出系统结构的好坏直接影响到计算机系统的性能，不仅仅是其输入输出的速度，而且还有其兼容性、扩展性、综合处理能力和性能价格比。

本章首先简要地叙述输入输出系统的发展过程，然后阐述输入输出系统的结构和设计。总线结构是 I/O 系统的重要组成部分；而在 I/O 系统中，输入输出的控制和设备的管理都要用到中断系统。所以，我们先集中地在 §2 叙述总线结构，在 §3 叙述中断系统。而后，在此基础上以 IBM360/370 的通道为例讲述通道型的 I/O 处理机。接着以 CDC-CYBER 系统的外围处理机为例讲述具有外围处理机结构的 I/O 系统。

§1 输入输出系统的发展过程

输入输出系统的发展大致可分为三个阶段，或是说可以分为以下三种方式：

1. 程序控制输入输出的方式

由第一章图 1.14 可知，早期的计算机是以 CPU 为中心，外部设备由 CPU 直接管理；输入输出操作完全由 CPU 控制，由 CPU 执行其启动、控制和停止；输入、输出设备与主存的信息交换需经过运算器。最早的一种方式是在进行输入输出操作期间，停止 CPU 的运算，直至数据交换完毕。

为了使用某个外部设备，CPU 需要与该设备进行以下联系：CPU 需选定和指明所要用的设备；CPU 需发出控制信号使该设备与运算器相连（直接相连或通过接口缓冲寄存器相连）；CPU 需发出该设备能“理解”的控制命令去启动该设备，使它能接受（对应输出操

作)或发送(对应输入操作)信息;该设备还需能在接受或发送每个信息元(如一个字符)时发送相应控制(状态)信号告知 CPU;由于信息的输入、输出一般是成块进行,所以,CPU 还需能检测或判定信息块的终点。由于输入、输出设备的信息表示(可能是二~十进制)可能和 CPU 内的信息表示(大多是二进制)不同,以及输入、输出设备与 CPU 之间的传送方式(如串行传送或是并串行传送)和 CPU 与主存之间的传送方式(都是并行传送)的不同,还需由 CPU 进行信息的变换、装配或拆卸。另外,CPU 还需能递增主存中存输入、输出信息块区域(称 I/O 缓冲区)的地址,以便能逐字(若主存与 CPU 之间是按字传送)存取;同时还需能递减信息块长度计数器(可能是存在通用寄存器内,也可能是存在主存内)的值。

上述这些操作是使用 I/O 设备时所必须的,至今基本上仍然如此。可以说,I/O 系统结构发展中的一个重要方面就是如何改进这些操作的分配,使它们分布实现,而不是如程序控制输入输出方式那样是集中的由 CPU 实现。

对于程序控制输入输出方式,虽然输入、输出指令的格式和功能可能不同,然而,这些操作总是由一段程序控制,而且往往是每输入输出一个信息元就需循环一次。更由于程序的进行要等待由慢速输入输出设备回授来的状态信息,致使计算机系统的使用效率非常低。CPU 与外部设备不能同时工作,各台外部设备之间也不能同时工作,计算机系统往往是只能有一部分在工作,资源得不到充分的利用,CPU 的高速运算能力远远得不到发挥。

之后,随着中断概念的出现,并将它引用于输入输出过程。例如,程序要进行打印时,执行一条打印指令去启动打印机之后,程序可不必空等由打印机返回的状态信息,可转去执行别的运算。等到打印机作好准备,发出中断请求,CPU 再响应中断,将信息送入打印机。接着,CPU 又可继续执行自己的工作。这样,增加简单的中断环节就可使 CPU 与输入输出能在一定程度上并行工作,且由于启动一种设备到该设备发中断请求的时间较长,足以启动另外的设备,从而可实现多种外部设备的同时工作。

由于输入输出设备的工作速度取决于机械动作,这不仅是使得它的信息传送速率远比 CPU 低,而且输入、输出设备与 CPU 之间只能是异步联系,再加上前述信息表示方式的不同,这就要求在 CPU 与各个 I/O 设备之间需要设置专用接口。其基本原理图如图 4.1 所示。

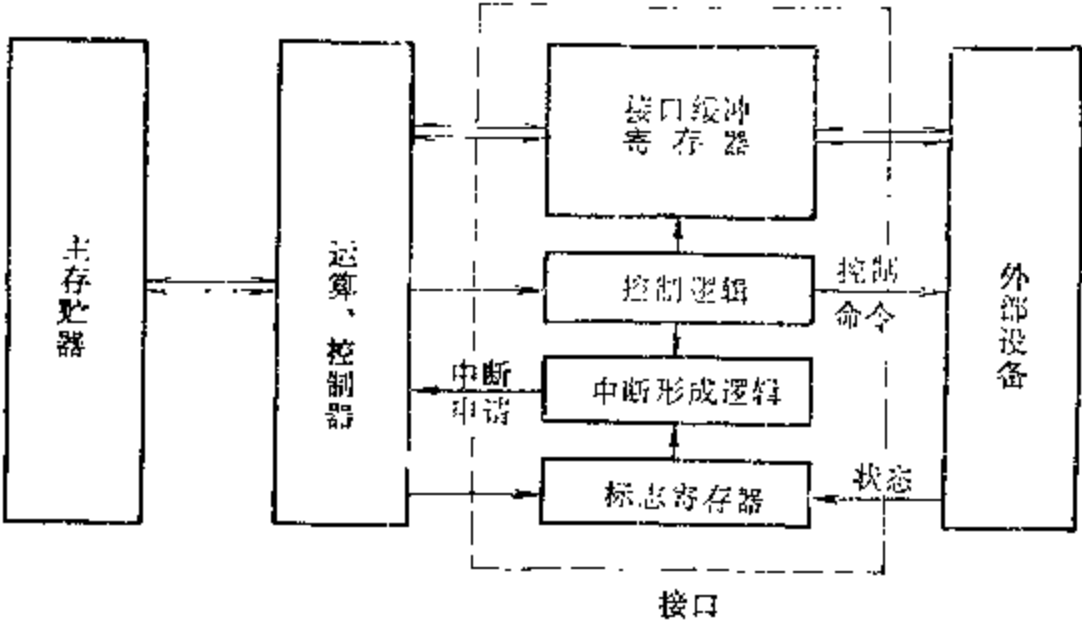


图 4.1 原始 I/O 接口原理图

早期机器的 I/O 接口是放在处理机内的，其控制逻辑也很简单，而且，处理机需为每种外部设备，甚至每个设备都配置相应的接口。这显然严重地限制了机器所接外设的可扩展性及兼容性。由于 CPU、主存的传送速率远大于 I/O 设备的，如主存的频宽（即单位时间所能存、取的字数）约为磁盘数据传送速率的 5 倍，约为磁带数据传送速率的 25 倍，约为行式打印机所需数据速率的 100~200 倍，约为纸带光电输入机数据传送速率的 5000 倍。因此，很快地就把上述专用接口方式改进为 CPU 经公用 I/O 总线再与多台外设相连的方式，当然要求 I/O 总线的数据流量需能满足多台设备同时工作的需要。可以说，接口方式改进是 I/O 系统结构发展的另一个主要方面。

2. DMA 方式

虽然采用中断技术可以在一定程度上实现 CPU 与外部设备以及多种外部设备间的并行工作，但随着外部设备种类和数量的增多，会造成中断次数过于频繁，而耗费了大量的 CPU 时间。另外，在程序控制输入输出方式中，信息是从内存经运算器与外部设备交往，如改进为在主存和外部设备之间设置直接通路及相应的控制逻辑，使外部设备能直接与主存相连，就能显著减轻 CPU 的负担。这就是所谓 DMA（直接存储器访问）方式，其原理框图如图 4.2 所示。

尽管有的机器的 DMA 是为某个快速外设（如磁盘）所专用，但大多数机器的 DMA 是通用的，如图 4.2 所示。DMA 使 I/O 总线可绕过处理机与主存总线直接相连。DMA 内有数据缓冲寄存器、设备地址寄存器、主存地址寄存（计数）器、数据计数器和其它一些控制线路。进行数据传送前，先由 CPU 执行一条或两条输入输出指令以完成往设备地址寄存器置入设备号，启动该设备；还把用于数据传送的主存区域的起始地址置入主存地址计数器，并往数据计数器置入所需传送的字数。之后，CPU 仍有可能继续执行其它程序。

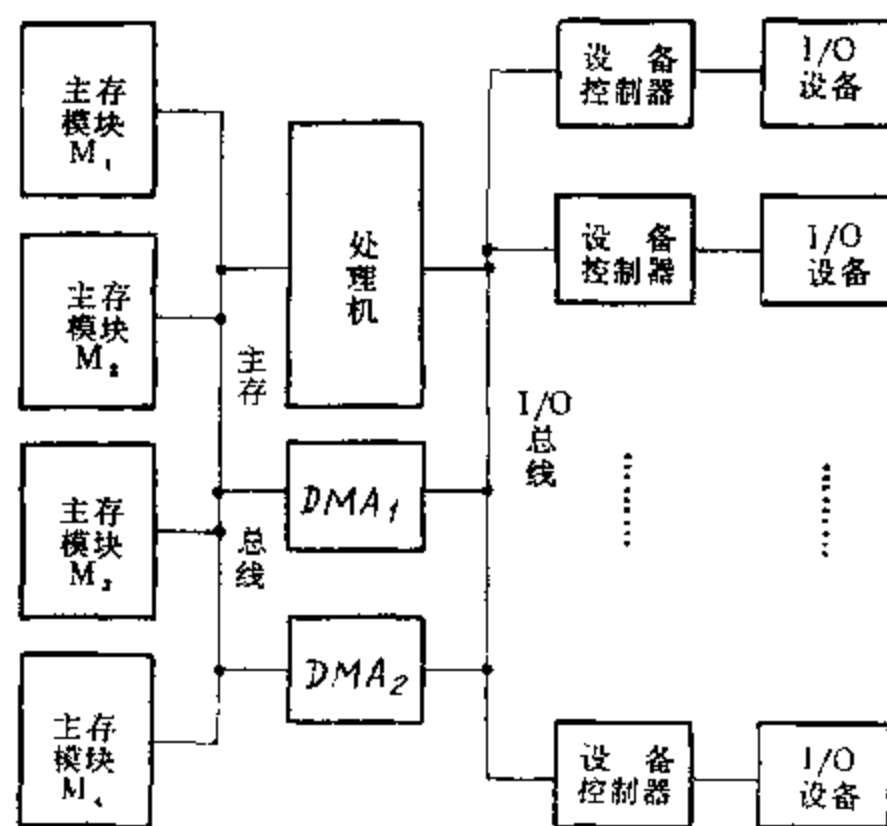


图 4.2 具有 I/O 总线和 DMA 方式的 I/O 系统

对于 DMA 成组传送方式，当输入输出设备准备好发送或接收数据时，它对 CPU 发 DMA 请求，使输入输出设备直接与主存交换数据。在送入或取出一个数据字后，输入输出设备给 DMA 内的主存地址寄存器地址加“1”，数据计数器减“1”。当数据计数器值成为“0”时，本次输入输出操作结束，输入输出设备发一中断信号告知 CPU。然而，有的机器则是在 DMA 成组传送期间停止 CPU 对主存的访问，这会使 CPU 因不能由主存取得指令或数据而停止运算。前面讲过，主存的频宽至少比 I/O 设备的数据传送速率高好几倍，所以，在 DMA 成组传送期间，主存并不是满负荷工作的，尤其是采用如图 4.2 所示的多体并行主存结构（在第五章还要详述）时，主存是有可能同时满足 DMA 和 CPU 的访存需要。因此，大多数机器是采用所谓 DMA 挪用主存周期方式。对它，在 DMA 传送一、二个字后

就可把主存的使用权送给 CPU, 使得有可能实现 CPU 的运算和该设备的操作重迭进行, 这些, 大家在计算机原理课中都已学过了。

DMA 方式的出现, 不仅使得 CPU 与外部设备有可能并行操作, 而且把输入输出过程中与主存交换信息的部分操作与控制交给了输入输出系统, 进一步简化了 CPU 对输入输出的控制。

3. I/O 处理机方式

虽然 DMA 方式简化了 CPU 对数据传送的控制; 然而, 对 I/O 设备的管理和不少操作仍需 CPU 承担, 而且信息的变换、装配或拆卸和数码校验等都得由 CPU 实现。随着分时系统的发展和任务数目的增多以及每台机器所接 I/O 设备数目的增加, I/O 系统的负荷亦日益加重; 为了使 CPU 能摆脱用于管理、控制 I/O 系统的沉重负担, 在六十年代初就出现了 I/O 处理机方式, 使机器具有如第一章图 1.15 所示的那种系统结构。I/O 处理机几乎把控制输入输出操作及其信息传送的所有功能, 从 CPU 那里接管过来, 独立出去。

I/O 处理机方式有所谓通道方式和外围处理机方式二种。IBM360/370 所用的是通道方式。这种“通道”是“处理机”, 它也有指令, 并能构成程序。通道通过通道指令对外部设备进行控制, 发出各种诸如读、写等命令, 给出主存地址及交换的字数。通道指令靠其链接功能构成存在主存内的通道程序, 与 CPU 程序并行工作。通道还能代替 CPU 对多个设备的信息传输进行分时管理, 并在主存和外设交换信息过程中实现字与字节之间的装配与拆卸。为了复执和诊断, 通道还能向 CPU 报告设备和设备控制器的状态和对状态的分析, 并能对输入输出系统出现的各种情况进行处理。

这样, 与外设进行一次信息交换的主要过程就成为: 当目态程序运行到一条要求与外设交换信息的访管指令时, CPU 进入管态, 由输入输出管理程序按要求形成通道程序, 并在启动通道后退出管态, 返回目态程序继续运行; 而被启动的通道按通道程序组织输入输出的信息交换, 一直进行到通道程序所要求的信息交换全部完成才向 CPU 发出中断; CPU 在响应中断后再次进入管态进行必要的登记处理, 又返回目态程序。因此一个外部设备的操作不论要发多少个命令, 也不论交换字数有多少, 只需二次转管, 从而大大减少对目态程序的干扰, 显著提高 CPU 运算和外设操作的重迭程度。当然, 更能使多种、多台外设同时并行工作。

由于通道对输入输出设备实行统一管理, 便于用户增加或更改外设, 通道通过通道总线的标准接口与外设联系, 有利于外设的扩展和更新。

一个计算机系统可以按需要接有多个通道, 每个通道各有其通道程序。

可以看出, 通道结构的出现使得原由 CPU 执行的 I/O 操作分散到一台或多台 I/O 处理机中去。我们在 §4 将详细讲述通道的结构和工作过程。

我们在这里所讲的“通道”和一般小型机所讲的“通道”具有不同的含义, 虽然在名词上往往是混用的。后者并不具有 I/O 处理机的特征, 它本身并不具有, 也不能运行通道程序; 它一般只具有前述程序控制输入输出和 DMA 的功能。

虽然 IBM360/370 的通道具有处理机的特征, 然而它只具有较窄的、面向外设控制和数据传送的指令系统, 它的程序是存在和 CPU 公用主存内, 通道内部只有用于数据缓冲的小容量存储器。这样, 输入输出过程中的码制变换(如二—十进制到二进制的变换, ASCII 码到二进制码的变换等), 整个数据块的错误检测和纠错以及文件的记录格式的变换等仍然

需由 CPU 实现。

与 IBM 发展通道型 I/O 处理机的同时, CDC 和 Burroughs 公司则是发展所谓外围处理机(或称前端机)的 I/O 处理机。外围处理机的结构更接近于一般处理机,或者就是采用已有的小型通用机。它既能完成通道的那些功能,还能完成上述码制变换、整个数据块的错误检测和纠错、格式变换等等运算和操作,从而能够进一步简化 CPU 对 I/O 系统的控制和使用。不过,就控制输入、输出过程的效率来讲,外围处理机可能没有专用性更强的通道高;而且,从当时看,如果是用分立元件去构成,那通道的价格要比外围处理机的低。关于外围处理机型的 I/O 系统以及它和通道型 I/O 系统的比较,我们在本章的末尾还要讲述。

我们在前面提过, I/O 系统的复杂化是由于分时系统的发展,是由于需要有更多的用户共享昂贵的 CPU 和主存,才能降低每个用户使用机器的费用。这种情况从六十年代开始一直持续到七十年代;然而,进入 LSI、VLSI 取得巨大进展的八十年代,情况已经发生了变化。首先是 CPU 的价格在整个计算机系统的价格中所占比例日趋变小,到了七十年代中期就已经比 I/O 系统所占的比例小,现在更是如此;因此,从更多用户分享 CPU 资源出发的分时系统思路已日益受到人们的怀疑。就是主存价格所占的比例也在降低,因此从分享主存资源的出发点也逐渐得不到更多的重视。尤其是在微处理机取得巨大进展的今天,单用户处理机在经济上和技术上的合理性有可能被更多的人所承认。

在这样的情况下, I/O 系统将向何处发展呢?当然,我们讲的发展单用户处理机主要不是指的又回到最初那种“孤单”的单用户处理机,而是指的由多台单用户处理机共享信息资源的计算机系统,即我们在上一章讲到过的分布处理系统。共享信息和共享 CPU、主存设备在概念上是不同的,共享信息指的是计算机的构成是如何使各个用户能共享巨大主数据库内的信息,并能在各个用户的处理机上对由主数据库调来的信息,按各个用户的需要进行高效率的交互式处理。对这种计算机系统,其 I/O 系统的设计要求和实现手段是会与目前计算机系统不同的;看来,有一点可以肯定,即在 I/O 系统上也会更多地采用基于微处理器的分布处理技术。上一章讲过的智能终端以及各种智能外设控制器(即在外设控制器内设置微处理器),尤其是与数据库磁盘相联的智能控制器,必将会得到迅速的发展。

由图 1.15 和图 4.2 都可看出,通道总线、I/O 总线是 I/O 系统的关键部分,它既要能传送数据信息,又要能传送控制信息;而且,还要能处理好 I/O 设备与处理机的联系以及如何使多台 I/O 设备能够交叉地、正确地经这些总线传送信息。下面我们集中地来讲述总线结构。

§ 2 总线结构

计算机系统结构设计的一个重要方面是如何按照功能要求安排好计算机系统内各个寄存器、功能块、部件和子系统间的互连。根据连接距离的不同,各种互连大致可分三类。

第一类是同一部件内的互连,例如,寄存器之间、寄存器与运算部件之间的数据传送和控制信息传送,它们的距离很短,但往往要求很高的速度,它们的控制简单,是大家所熟悉的。第二类是计算机系统(主要指的是单处理机系统)内部的互连。例如,中央处理机、存储器与通道、I/O 设备的互连等,如前述的主存总线和 I/O、通道总线。连接距离比第一类长,其传送速率可以略低,但差异很大。前面讲过,从磁盘机到光电输入机,它们的传送速

率差别可以达 1000 倍以上。第三类是在多处理机或多机系统中的互连，它们之间的距离比第二类的还远，对于计算机网甚至可达数千公里。它们同样有速度差异的问题，而且可靠性和成本显得更为突出。

我们在本节着重讲述第二类互连的 I/O、通道总线，然而其基本概念也适用于第三类。

计算机很早就采用总线结构。I/O 系统的总线结构比较复杂，它的设计好坏，对 I/O 系统的性能影响较大。各种总线都可以通过结构格式和基本参数来表征。本节对诸如总线的类型、控制方式、通讯技术、数据宽度和总线线数等加以讨论。

计算机系统结构设计者的任务就是根据系统的功能和其它要求，全面平衡，努力设计出低成本高效率的总线结构。

2.1 总线的类型

总线是由源部件到目的部件传送信息的一组互连线。它可以是单向传输的，即只能向一个方向传送，例如，读卡机仅给计算机提供数据，穿孔机、行打机仅接收计算机来的数据。它也可以是双向传输的，例如，磁带、磁盘、用户终端与主机的联系，系统与系统之间的联系等等。计算机系统中单向传输用得较少，大量应用是双向传输。双向传输又可分为所谓半双向和全双向。前者允许在两个相反方向传送，但同时只能向其中一个方向传送；后者允许同时在二个方向传送。全双向的速度快，但造价高，结构复杂，很少应用；如果不加说明，双向传输就是指的半双向传输。

总线按其用法可以分成二种类型：专用的和非专用的。

2.1-1 专用总线

只赋以一种功能或只实现一对物理部件间连接的总线，我们都称之为专用总线。例如，图 4.3(a)的二根总线既是分别只为处理机提供程序和数据，又是只实现一对部件间的连接，从这二方面看都是专用的。然而，若从程序总线既传送地址又传送操作数这点来看，则它又可看作只是物理上的，而不是功能上的专用。同样，如果在程序总线上接有多个存放程序的存贮器模块，那么它的总线将只在功能（提供程序）上，而不是物理上的专用。但是，如图 4.3(b)那样，功能上专用于地址和专用于操作数的二根总线，物理上显然是非专用的。总之，从物理连接上来定义专用是明显、确切的；而从功能上来定义专用，则取决于“功能”的含义，不明显，因此，如果不加说明，则专用都是指只连接一对物理部件来定义的。

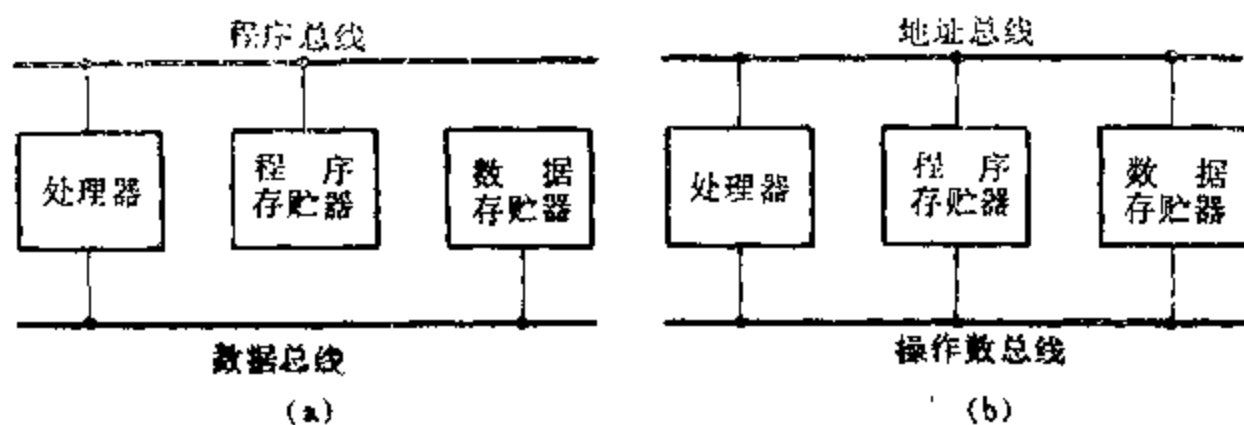


图 4.3 功能上的专用和物理上的专用

专用总线的基本优点是具有高的流量，多个部件可以同时发送或接收信息，几乎不会出现总线争用的现象。如果 n 个部件需要用双向专用总线在所有可能路径互连的话，则需用 $n \cdot (n-1)/2$ 条总线。图 4.4 的实线是 $n=4$ 的情况。这种专用总线控制直捷简单，源和目的是唯一的，从而不用指明。任何连线的失效仅影响连到该线的二个部件之间的通讯，而且，如果它们能经系统的其它部件传递信息的话，也仍然可以进行通讯。例如，图 4.4 中，在 A 与 C 之间的总线失效时，还可以经 B 或 D 传递，从而增加了系统的可靠性。

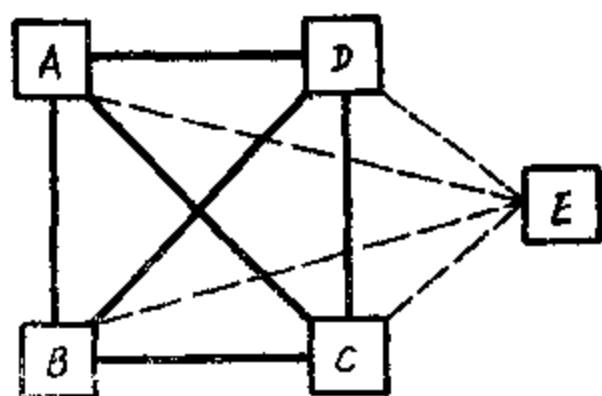


图 4.4 所有部件之间用专用总线单独互连

专用总线的主要缺点是总线数目多，成本高。虽然总线不完成逻辑运算，但通常包括控制用开关、信号驱动和整形电路。如图 4.4 这样的情况，专用总线的个数是近似地随部件数的平方而增加的，这就使得转接头和总线用集成电路也随部件数的增加而迅速增多。如果总线较长，则所用导线或电缆的成本也是不可忽视的，如果采用设置双套总线来进一步提高可靠性，那成本更是会成倍地增加。另一个问题是专用总线的时间利用率往往很低。专用总线也不利于系统的模块化，增加一个部件要增加许多新的接口和连线（见图 4.4 虚线所示）。在一般 I/O 系统的各个设备之间，专用总线只用于实现某个设备（部件）仅与另一个设备（部件）的相连，如图 4.2 中的 I/O 设备与其控制器的相连。如果不加说明，则我们所讲的 I/O 总线都是指的下面要讲的非专用总线。本节内容也是围绕它来讲述的。

2.1-2 非专用总线

非专用总线可以被多种功能或多个部件所共享。每个部件能通过共享总线与接在总线上的其它部件相联。但在同一时候，却只允许二个部件经共享总线相联，其它部件之间的相联要在别的时间进行，因此也可称为分时共享总线。

有的微、小型机，整机只有一根共享总线（单总线结构），如图 4.5(a) 所示，它不论在功能上还是物理上，都不是专用的，而且既是 I/O 总线又是主存总线。图 4.5(b) 是为了防止单点失效的多（三）重非专用共享总线，用于提高可靠性。有些小型机和大、中型机则是让 I/O 总线和主存总线分开，一种型式如图 4.5(c) 所示。这有助于解决好 I/O 设备和 CPU、主存之间传送速度的差异。图 4.5(d) 是共享总线用在远距离通讯中的一个例子。

单总线结构能经同一总线实现 I/O 设备之间的直接联系和 I/O 设备与主存之间的直接联系。由于可使 I/O 设备作为主存单元来统一编址，如 PDP-11 那样。所以，处理机不需要有专用的 I/O 指令，或是说，所有与主存有关的，功能很强的指令都可用于 I/O 操作，这是单总线结构的突出优点。另外，机器的所有部件和设备都具有相同的总线接口，有助于简化和统一接口的设计。

然而，对于单总线结构，总线流量和总线长度的矛盾很突出。这是因为，它的流量需满足 CPU 与主存之间所需的信息流量，对于 1 微秒单体主存来讲，就需高达每秒一百万字，然而，它的长度却需长到能把机器的所有部件（包括外设控制器等）都联起来。要采用速度又快、长度又长的总线，其造价要提高很多，甚至难以实现。所以，当以单总线结构著称的 PDP-11 进一步提高其主存速度，并增设超高速缓冲器（Cache）时，也不得不把主存总线

与 I/O 总线分开,如第一章图 1.21(b)所示; 甚至还需把快速外设与慢速外设的 I/O 总线分开如图 1.21(c) 所示。总之,对非专用总线的设计,要处理好流量、长度和造价之间的关系。

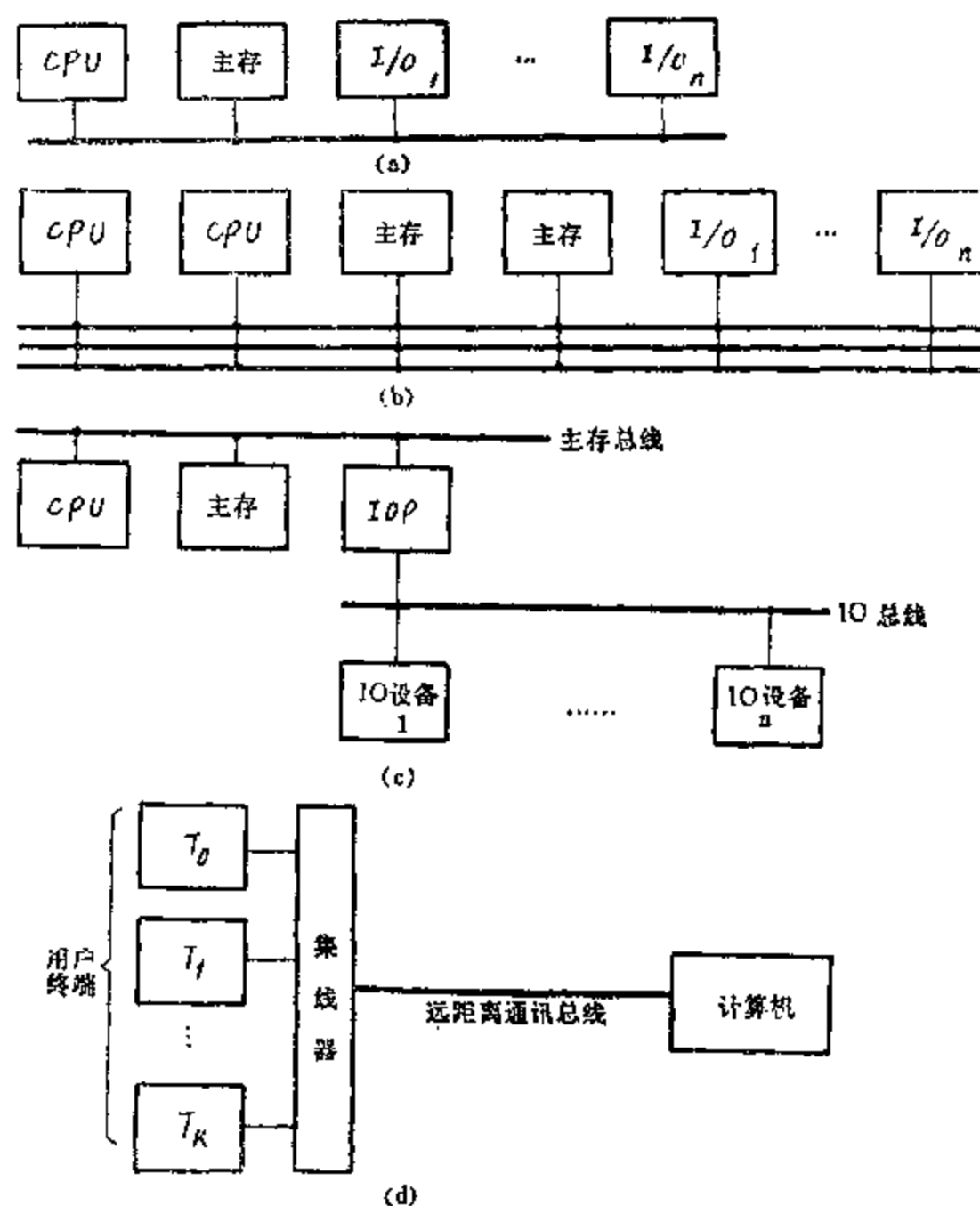


图 4.5 共享总线举例

纵横开关矩阵是互连一类部件(如存贮器)中的任何部件到另一类部件(如处理器)中的任何部件的非专用总线结构。它可同时实现所有部件之间的相联,但总线的使用率往往较低。这种纵横开关连接也可用于 I/O 处理机和 I/O 设备之间。如图 4.6 所示为 m 个部件与另外 n 个部件互连的交叉开关。其纵横交点都有一个开关,逻辑上可以从任何水平通路通到任何垂直通路。任何时候,每行每列中至多只能有一个开关可以接通,如果 $K = \min\{m, n\}$,则行中任何 K 个相异的部件可以同时连到列中任何 K 个相异的部件。就是说它允许直至 K 个数据传送同时进行,这就大大减少了争用总线的现象,提高了数据在系统中的传送能力。当然,这种开关的控制复杂,且开关的数目随部件的数目成平方比增加,而一个开关的失效会使相应部件之间没有另外的通路。

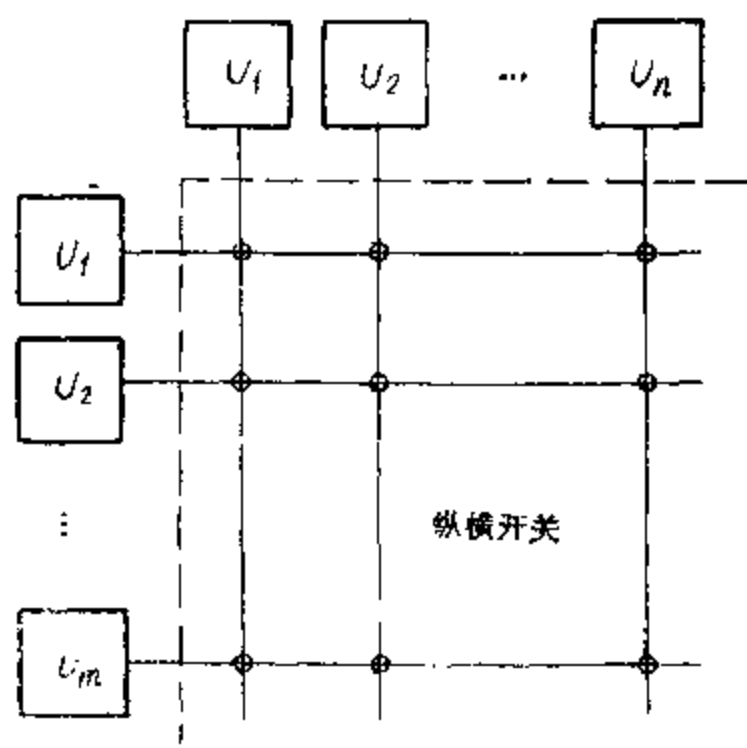


图 4.6 二组不同类型部件的交叉开关连接

非专用总线的主要优点是可降低成本，这是由于总线的数目可以比专用总线方式大大减少。另一个特点是它有助于实现模块化，具有较大的灵活性。在非专用总线上增加部件要容易得多，避免了电缆、接口和驱动电路等的激增。而且，如图 4.5(b) 所示，比较容易设置多重总线来提高总线的带宽和可靠性，使故障弱化。

非专用总线的缺点是不允许多对部件经同一总线同时通讯，从而会出现争用总线，这会使未获得总线控制的部件处于等待而降低了效率。如果处理不当，总线有可能成为机器速度的瓶颈，对单总线结构尤其如此。其次，系统对共享总线的失效特别敏感，它往往可以导致系统中的整个联系瘫痪。

由于可能有多个设备或部件同时申请使用总线，就得有总线控制机构来控制，保证在同一时间内只能有一个申请者取得对总线的使用权。控制机构还要控制好源和目的的连接，并使信息在源和目的之间正确传送。

2.2 总线的控制方式

本节就是讨论如何控制总线的使用权，如何确定各个申请者享用总线的优先次序。当有多个部件同时申请总线时，必须按照某种优先次序，保证在给定时间内只有一个高优先级的部件能使用该总线。

根据总线控制器的位置，控制方式可以分为集中式和分布式两类。总线控制逻辑基本集中在一处，不论是在连接到总线的部件中，还是在单独的硬件中，都称为集中式总线控制。而总线控制逻辑分散在连到总线的各个部件时，就称为分布式总线控制。

总线应该包括有数据线、地址线和控制线。数据线用于传送数据，地址线用于传送目的和源的地址（设备、部件号）。可以使数据和地址分时使用数据线，以减少总线线数，降低成本，但这会影响信息传送速度。控制线用来传送为控制总线所必需的控制信号、状态信息以及指明数据线上的信息类型等。虽然控制线的功能也可以通过编码方式经数据线传送以减少总线的线数，但这会影响到总线的分配速度。

优先次序的确定可以有三种方式：串行链接、定时查询和独立请求。当然也可以是它们的结合。其选择取决于控制线数目、总线分配速度、灵活性、可靠性等因素的综合权衡。

2.2-1 集中式总线控制

图 4.7 表示集中式的串行链接方式。所有部件是经公共的“总线请求”线发出申请。当“总线请求”被激励时，表明有一个或多个部件请求使用总线。这时，只有当“总线忙”信号未建立时，“总线请求”才能被总线控制器响应，送出“总线可用”回答信号，它串行地

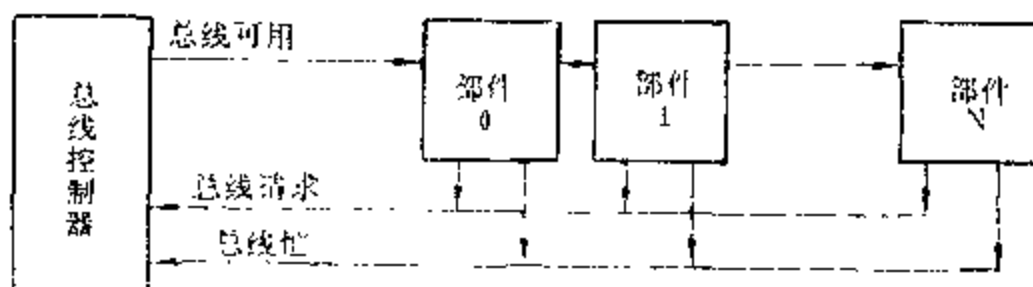


图 4.7 集中式总线控制：串行链接

通过每个部件。如果某个部件接收到“总线可用”信号，但并没发过“总线请求”时，则将该信号继续送到下一个部件去；如果该部件接到“总线可用”信号并且发出过“总线请求”时，

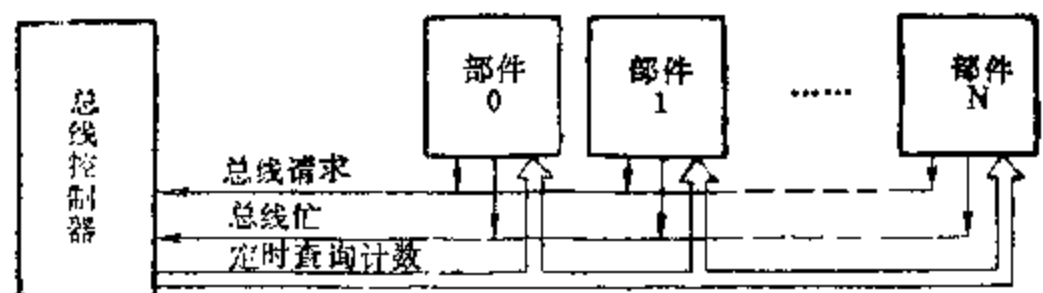
则“总线可用”信号停止传送。该部件建立“总线忙”，并去除其“总线请求”，之后即可进行数据的传送。在数据传送期间，“总线忙”维持“总线可用”的建立。完成传送后，部件去除“总线忙”信号，“总线可用”随之去除。其后，当“总线请求”再次建立时，就开始新的总线分配过程。

可以看出，同时有二个以上部件请求总线时，离总线控制器近的部件将获得总线。就是说，优先次序完全由“总线可用”线所接部件的物理位置来决定，离总线控制器越近的部件其优先级越高。

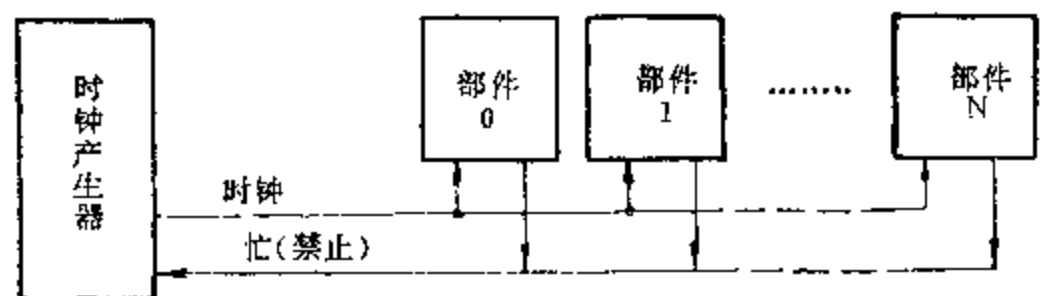
串行链接的优点是选择算法简单，控制线少，且不取决于部件的数量；部件的增加容易，只需简单地把它连到总线上；由于逻辑简单，也就容易提高其可靠性。缺点是串行链接方式对“总线可用”线及其有关电路的失效很敏感，如果部件*i*不能正确传送“总线可用”信号，则在*i*之后的所有部件将永远得不到总线。由于优先级是线连固定的，不可能被程序改变，因此，如果离总线控制器近的部件出现频繁的总线请求时，离总线控制器远的部件就很难取得对总线的使用权。由于“总线可用”信号必须顺序、脉动地通过各个部件，这要降低总线分配的速度。由于受总线长度的影响，增加、去除或移动部件也要受到限制。

图4.8(a)为采用统一计数器的集中式定时查询系统。总线上的每个部件可以通过“总线请求”线发出请求，总线控制器收到请求后，让计数器开始计数，定时地查询各个部件以确定是谁发的请求。当查询线上的计数值与发出请求的部件号一致时该部件就建立“总线忙”，控制器中止查询，直至该部件完成传送，去除“总线忙”。这种计数可以从“0”开始，也可以从中止点继续。如果每次从“0”开始，部件优先级的排序类似于串行链接，而若从中止点继续则是一种循环方法，为所有部件提供相同的使用总线的机会。采用何种计数办法取决于优先级的安排。由于计数器的初始计数值可以由程序置定，所以优先次序也就可以被程序控制。这种定时查询不会出现在串行链接式中某个部件失效影响到其它部件对总线的使用的现象。然而这种方法的灵活性是以增加控制线线数为代价的，可以共享总线的部件数是受限于定时查询线的数量（编址能力），而且控制较为复杂。

图4.8(b)为使用局部计数器的集中式定时查询系统。它在每个部件中设置一个计数器，从而可以免除上述统一计数器法对增加部件数的限制。总线控制器仅为一个时钟信号产生器，它给所有部件的计数器计数。当计数值到达发出“总线请求”的部件号时，该部件建立“忙”信号，“忙”信号就去禁止时钟的发出。当部件完成传送时，去除“忙”信号。同样，如果计数是循环地连续，则总线分配是循环的；如果所有部件计数器都从“0”开始，则部件优先级受其编号次序决定。显然，可以通过改



(a) 采用统一计数器的定时查询



(b) 采用局部计数器的定时查询

图 4.8 集中式总线控制：定时查询

变赋予部件的编号来改变优先级。这种方式的主要缺点是会因时钟信号的倾斜而出错，因此被限制应用于小的低速系统中。它对干扰和时钟的失效特别敏感。

图 4.9 表示集中式的独立请求方式。共享总线的每个部件都有各自的一对“总线请求”和“总线准许”线。当部件请求使用总线时，送“总线请求”信号到总线控制器。由它确定哪个部件可使用总线，并立即通过相应的“总线准许”送回该部件，去除其请求，建立“总线已被分配”，以表明总线正忙。数据传送完后，部件去除“总线已被分配”。经总线控制器去除“总线准许”，并确定下一个可使用总线的部件。

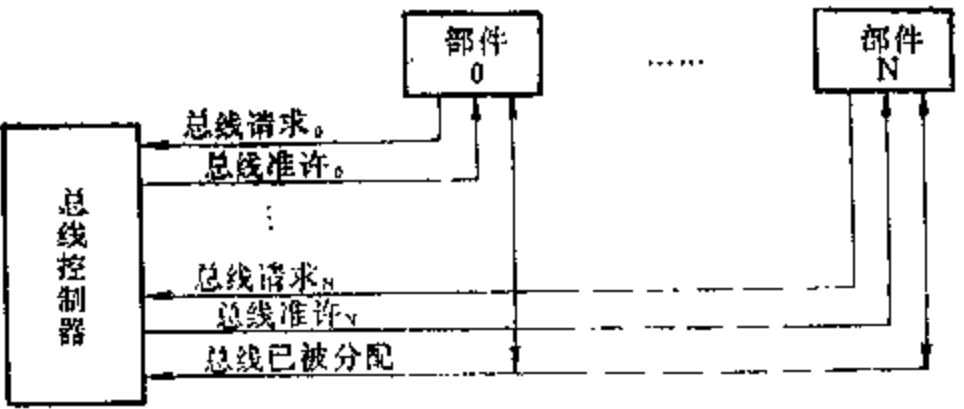


图 4.9 集中式总线控制：独立请求

独立请求方式的优点是总线分配的速度快，因为所有部件的总线请求可同时送到总线控制器，为分配总线所需的辅助操作时间比串行链接和定时查询的都短。这种方式的另一优点是控制器可以灵活地确定下一个使用总线的部件，可以使用程序可控的预定方式或自适应方式，也可采用循环方式或混用的方式。它也能方便地不响应来自已知失效或可能失效的部件发出的请求。缺点是控制线数量过大。为控制 N 个设备必须有 $2N$ 根“请求”和“准许”线。相比之下，串行链接只需 2 根，定时查询只需约 $\log_2 N$ 根线。另外，其总线控制器要复杂得多。

上述这三种方式也可用于分布式控制，下一小节分别来讨论。

2.2-2 分布式总线控制

图 4.10 表示分布式的串行链接方式。图 4.10(a) 是将集中式串行链接省去“总线忙”线并把“总线请求”线直接连到“总线可用”线来构成的。只要“总线可用”线为低电平，就允许部件请求使用总线；但只要有一个部件建立总线请求，就使“总线可用”线为高电平，不请求总线的部件让该电平通过，若遇到的是申请总线的部件就不再往下传送，并维持“总线请求”的建立，直至该部件结束使用总线，去除其“总线请求”。如果此时其它部件没有请求，就使“总线可用”线回复为低电平。因此当有多个部件请求总线时，在同时请求总线

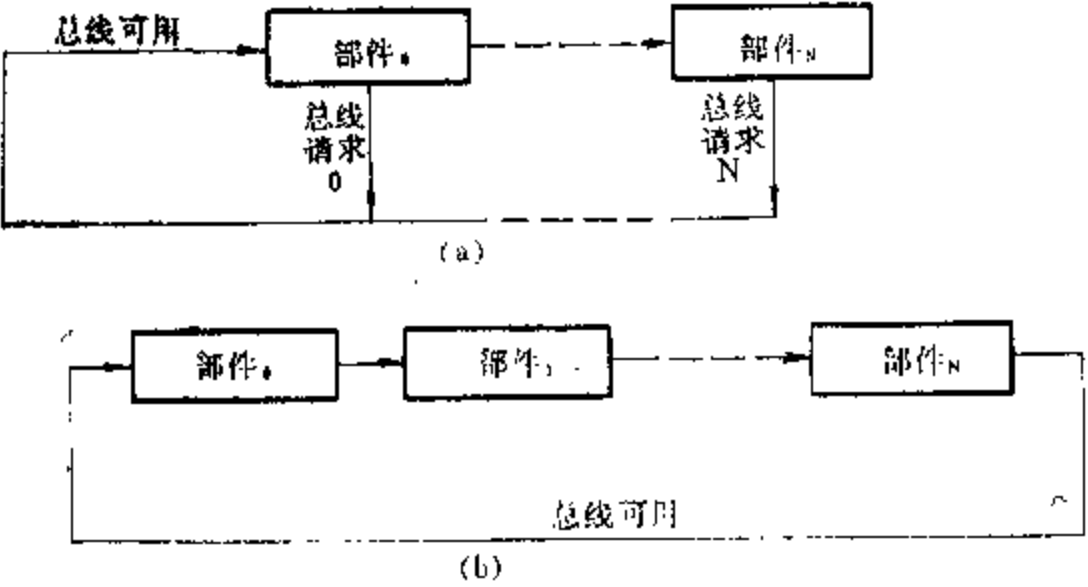


图 4.10 分布式总线控制：串行链接

的部件中最左的一个首先获得总线。而后，由于有其它部件请求总线，“总线可用”信号仍

为高，不管在此期间其左面的部件有否新的请求，“可用”信号继续由该部件向右传送，所以这种总线的分配是采用循环算法。

图 4.10(b)的分布式串行链接不是用“总线可用”线上静态电平的高低，而是用其线上电平的跳变来确定总线的状态。每当进入某个部件的“总线可用”线电平发生跳变，而该部件又请求使用总线时，就不让传送到下个部件的信号发生跳变；如果该部件没有申请使用总线，则让传送到下个部件的信号发生跳变。这样，在总线闲着时，“总线可用”信号的跳变总是沿串行链不断传送，只有当接收到“总线可用”线的电平跳变，而本身又是请求使用总线的部件时才停止这种信号不断跳变的现象，并让该部件取得总线的控制。当部件结束使用总线时，它又使送往下一个部件的“总线可用”线电平发生跳变。这种取决于信号跳变，而不是电平值本身的办法对于抗比前一种方法更敏感。上述二种分布式串行链接同样有着前述集中式串行链接对单点失效敏感和部件在链中所处位置的问题；虽然它们的总线分配算法简单，但很不灵活。

图 4.11 表示分布式的定时查询方式。当一个部件释放总线时，它把一个新的代码(地址或优先级)放在“定时查询代码”线上。如果代码与需要总线的另一个部件的编号(代码)相符，那个部件就用“总线接受”的建立来响应，此信号使原先部件去除“定时查询代码”和“总线可用”信号；接着现行部件去除其“总线接受”信号并开始使用总线。如果发出定时查询代码的部件没有能接到“总线接受”信号，就根据某种分配算法(循环的或优先级法)改变代码进行重试。这种办法要求系统一旦启动就得有一个部件占用着总线。分布式定时查询使用更多的硬件，但单个部件的失效不会影响总线的工作。

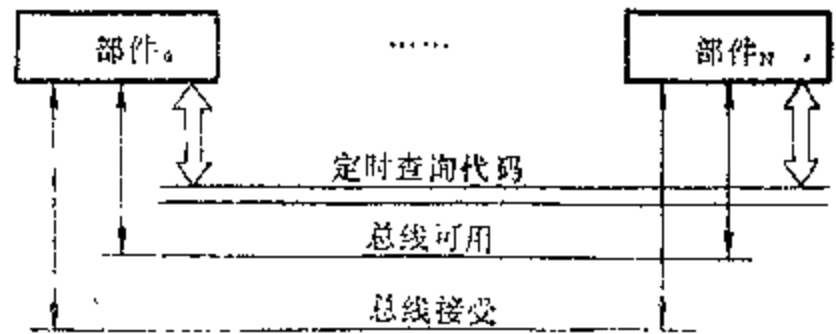


图 4.11 分布式总线控制：定时查询

图 4.12 表示分布式的独立请求方式，各个部件的“总线请求”线具有不同的优先级。需要总线的任何部件都可建立起它的总线请求。当现行部件通过去除“总线已被分配”信号来释放总线时，所有正请求使用总线的部件均按优先级别检测全部起作用的总线请求，以判定该由哪个部件使用总线。如果某个部件判定它的优先级最高，则通过建立“总线已被分配”信号来得到总线，并使所有其它部件发出的请求信号不起作用(如果采用循环算法就同时把下一个该使用总线的部件的优先级存起来)。分布式独立请求的每个部件中的优先判别逻辑比集中式独立请求的简单，但每个部件所接的线数要比集中式的多。时钟信号的倾斜问题限制它仅能用于小系统中，并且对于干扰和时钟失效特别敏感。



图4.12 分布式总线控制：独立请求

上面我们介绍了分布式总线控制的一些基本思路；然而，目前广泛应用的还是集中式控制。就集中式控制的三种方式来讲，虽然各有其优、缺点，但目前用得最广泛的，还是串行链接方式；主要原因就在于它简单，便于实现，总线控制器的结构和控制线的数目与所接部件的个数无关，而且各个部件的分配用接口以及对总线的申请和取得使用权方式都是一样的。至于其各个部件的优先级是线连固定的缺点，有的机器是通过设置具有不同优先级的多

根“总线可用”线(参看图4.7)来弥补。

上面讲了如何申请到总线的使用权。在获得使用权后,还需解决源和目的部件如何通过总线进行联系和通讯。

2.3 总线的通讯技术

当获得了总线的使用权后,还必须知道它是“源”还是“目的”,以及所传送信息的类型和操作的种类。信息类型究竟是命令、状态、中断等,还是数据,希望是部分或整个地隐的指明,或者是由操作种类一并指明。操作种类确定所需执行的功能,如输入、输出等等。只有在源和目的部件之间明确这些之后,才能开始实际的信息传送。信息在总线上的传送方法基本上可按同步和异步划分。

2.3-1 同步通讯

同步通讯时,二个部件通过总线传送信息是由定宽、定距的时标同步。时标可以由中央时标发生器发送到接在总线上的所有部件;也可以是每个部件各有其时标发生器,但它们都由中央时标发生器发出的时标同步。采用同步通讯,信号的传送速率高,且受总线长度的影响小,但它会因时钟在总线上的时滞而造成同步误差,而且时钟线上的干扰信号易于引起错误的同步。

为了提高可靠性,当然希望目的部件对数据是否已被接收、以及是否正确均能给出回答。如果同步时间片的宽度要宽到能为每个字的传送作出回答,则它必须是按接到总线上的最低速的部件来考虑,这就会使同步通讯的数据传送速率低于后面要讲的异步通讯的。一种解决办法是在正常时,目的部件不作回答,源部件也不等待回答信号;但如果发生错误则目的部件将在同步时间片过去之后,发回源部件一个出错信号,这样,就不会降低正常时总线的传送速率。但是这种办法中,源部件必须能保留已传送的,但未经证实和回答过的所有数据,以备重发之用。

2.3-2 异步通讯

由于I/O总线一般是为具有不同速度的许多I/O设备所共享,因此不适于采用同步通讯,而宜于采用异步通讯,异步通讯基本上可以分成单向控制和双向(请求/回答)控制两种。

单向控制指的是通讯过程只由目的或源部件中的一个控制;而双向控制是由源、目的双方共同控制。单向控制又有源控制和目的控制两种。

图4.13(a)为异步源控式单向控制通讯。源部件将数据放在“数据”总线上,并在控制总线上发“数据准备”信号。“数据准备”的时间关系取决于目的部件如何使用它。如果它在目的部件中作为接收数据的选通信号用,则延迟 t_1 是必须的,以防止“数据准备”信号在“数据”之前到达目的部件。但这使得需按源、目的间的不同距离调整 t_1 值,从而使得传送速率和距离有关。因此,不如让“数据准备”与“数据”同宽,但目的部件在装入前在内部要使“数据准备”信号延迟。 t_2 是为源部件输出寄存器再装入新的待发数据或是为总线进行重新分配所需的时间。

这种由源控制的单向控制通讯的主要优点是简单和高速;缺点是没有来自目的部件指明传送是否有效的回答信号,而且对具有不同速度的部件之间的通讯比较困难,效率也低。另

外,“数据准备”线上的干扰易被错认为是有效信号,虽然这可以通过适当的定时使出错机会减少,但会因此而降低了传送速率。

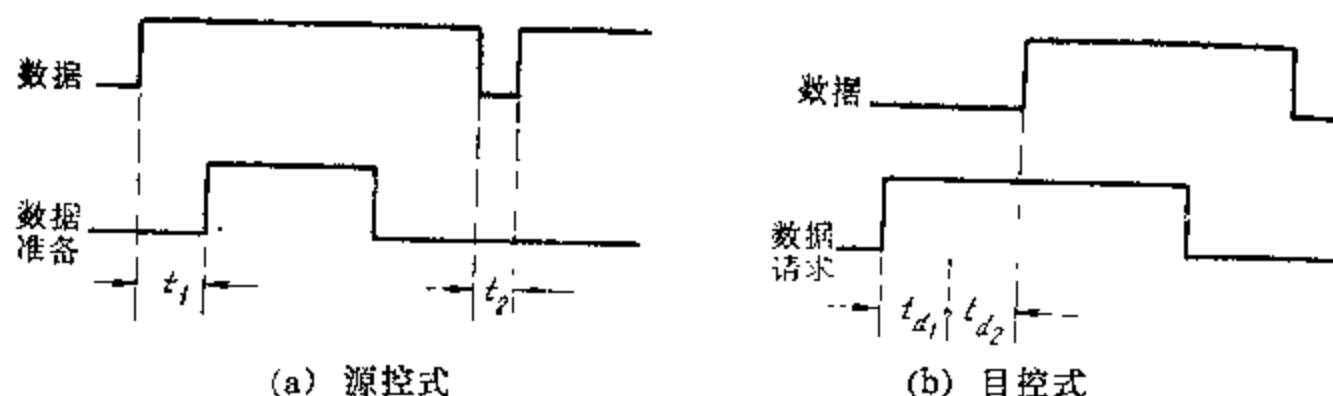


图 4.13 异步单向控制通讯

图4.13(b) 为受目的控制的单向控制通讯,以解决传送有效性校验的问题。由目的部件建立“数据请求”来使源部件把数据放在“数据”线上,在目的部件发出“数据请求”,需经二次传送延迟,即 $2t_d$ 后,数据才开始到达目的部件,并由它检验其有效性。如果有错,目的部件送“数据出错”信号来代替下一个“数据请求”。显然,总线传送速率随源、目的间距离的增加而下降,此外,开始时需有辅助操作时间以及目的部件在接收数据之后,需再加上错误校验时间才能发出下一个“数据请求”,这些也使传送速率进一步下降。

单向控制的缺点是未能提供数据传送完成的标志,从而可能造成错误,为此可采用异步请求/回答式双向控制。双向控制也有主从关系,以下以源为主来介绍。

图4.14(a) 为非互锁方式。源将数据放在总线上,稍迟,经 t_1 发出“数据准备”,目的部件在 $t_1 + t_{d1}$ 接收数据后发“数据接受”来响应,并返回到源部件在 $t_1 + t_{d1} + t_{d2}$ 去除原数据,而后再经过 t_2 时间把新的数据放在数据线上。如果目的部件发现数据有错,则发出“数据出错”代替“数据接受”。这既提供出错控制,也便于实现不同速度的部件之间的通讯;其代价是要降低传送速率,因为要经过二次总线延迟,并增加某些控制逻辑。如果总线传输延迟与通讯信号脉宽的比值不合适时,就可能出现下一个“数据准备”到达目的端时,上一个“数据接受”仍处于高电平,从而使得由此“数据准备”信号使“数据接受”线一直维持在高电平,如图 4.14(a)“数据接受”线上的虚线所示,造成错误。

显然,若使这下一个“数据准备”信号只能在上一个“数据接受”信号结束后才能发出,就能防止这种出错。图 4.14(b) 的互锁方式就能达

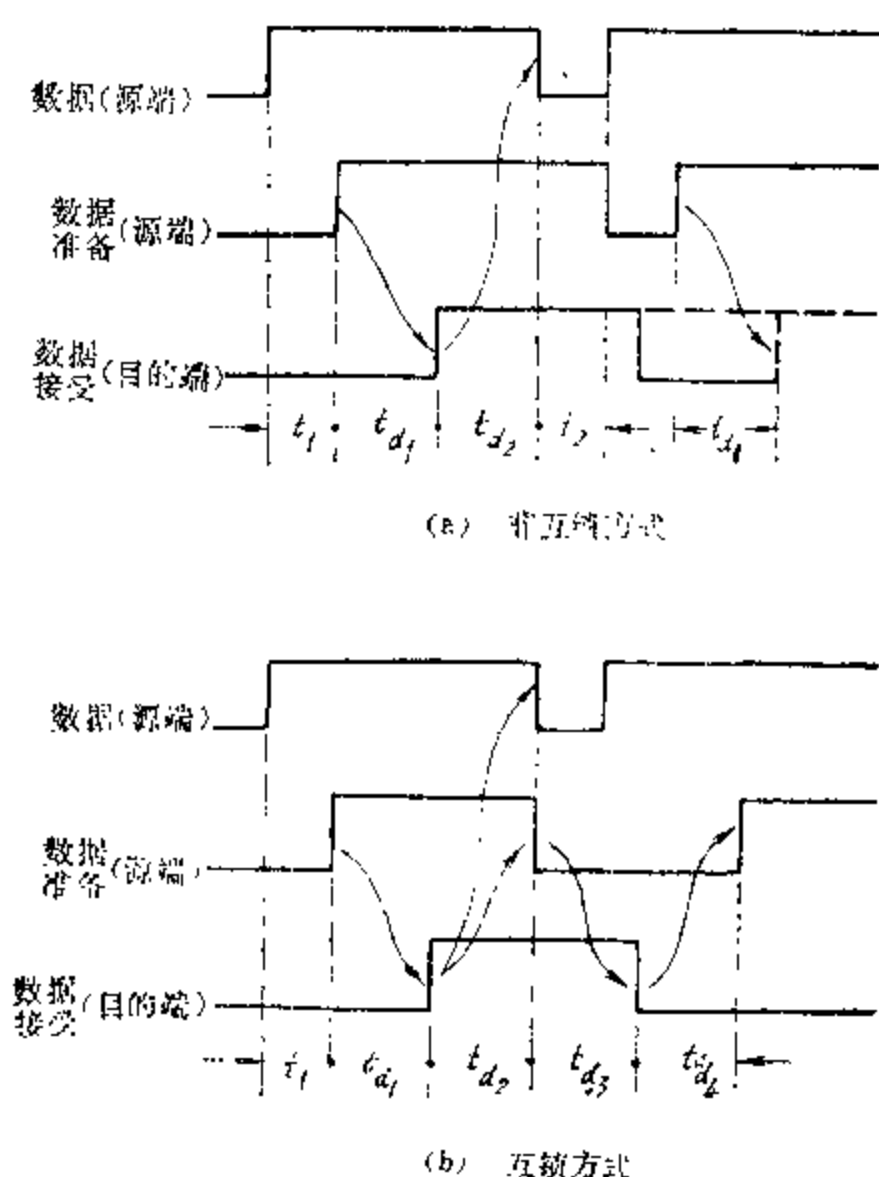


图 1.14 源控式异步双向控制通讯

到这点。对它，只有当“数据接受”的后沿返回到源端后，下一个“数据准备”信号才可发出。

注意，我们在图上所画的 t_{d1} 值是要大于一次总线延迟的时间。例如，对 t_{d1} 来讲，当“数据准备”信号前沿经总线延迟到达目的端，并把数据送入目的端的数据寄存器后，需再加上数据校验时间才能发回“数据接受”信号。

虽然异步双向互锁方式增加了信号沿总线来回传送的次数，也使控制硬件稍许复杂些，但它能适应各种 I/O 设备的不同速度，保证数据传送的正确性，因而是目前 I/O 总线中使用最广泛的一种方式。

2.4 数据宽度与总线线数

2.4-1 数据宽度

我们这里所讲的“数据宽度”指的是 I/O 设备取得 I/O 总线的使用权后所传送数据的宽度，在传送完这个宽度后，总线就重新对各申请者进行分配。采用何种数据宽度与总线上各设备的工作特点、所采用的总线控制方式和通讯技术有关。宽度的种类有单字、定长块、可变长块、单字加定长块和单字加可变长块等。

单字宽度不适合面向成块信息传送的设备（部件），如磁盘、磁鼓、磁带等。因为这些设备本身有较长的访问等待时间，而且在能使用总线前又往往需进行各种辅助操作；因此，若每传送一个字都需要化费访问等待和辅助操作时间，那必然会显著降低对这些设备的等效访问速度。另外，对于单字宽度，每传送一个字都要重新分配总线，这就得要求总线控制逻辑能高速地重新分配总线。这往往会妨碍采用合理的分配算法，或是不能充分利用总线本来可以提供的带宽。

采用定长块宽度可以增大总线带宽，简化控制，并可按整个信息块进行校验。但由于块的大小固定，当它与进行通讯的部件的信息块大小不相符，例如小于部件的信息块时，虽比仅单字传送的好些，但仍未能完全避免上述仅单字传送的那些缺点。而如果大于部件的信息块，则会浪费总线的带宽和缓冲器空间，同时也会超过所需时间，过长地束缚住经总线传送信息的二个部件，使之不能及时转入别的操作。

采用可变长块宽度，可按设备的特点动态地改变传送块的大小，显然比上述二种形式都要灵活。这可以使传送块的大小和部件的物理或逻辑要求相配，从而能更有效地利用总线的带宽，也使通讯的部件能全速工作。因为总线进行信息块传送前所花的辅助操作时间与单字传送中所花的时间一样；所以，可以根据对总线流量的要求和平均信息块的大小，在带宽和灵活性之间折衷权衡。例如，若让块的大小总是取成源部件的信息块大小就可不用指明块的大小，从而可减少辅助操作。这种方式的数据出错控制仍可以按字进行。

对于挂有速度较低，而优先级却较高的设备的总线，可以采用单字加定长块传送。这样，定长块的大小就不必选择过大（信息块超过定长块的部分可用单字处理），从而可减少在总线带宽、部件的缓冲器空间以及部件的可用能力方面的浪费。当然，若所要传送的信息块小于定长块的大小，但字数又不少时，设备或总线的利用率都会降低。

采用单字加可变长块传送，是最灵活、有效，但是最复杂且花钱最多的方案。当要求传送单字时，这个方案比之于只能成块传送的方案可省下原用于成块传送的不少起始辅助操作，而当要求成块传送时，块的大小又能调整到与部件和应用的要求相适应，从而优化了总线的使用。

2.4-2 总线的线数

总线需要有发送电路、接收电路、传输线(导线或电缆)、转接器(转换插头等)和电源等。这部分比起逻辑线路的成本高得多,而且转接器占去了系统中相当大的物理空间,往往是系统中不可靠的部分。总线的线数越多,则成本越高、干扰越大、可靠性越低、占用的物理空间也越大,当然传送速度和流量也越高。此外,总线的长度越长,成本就越高,干扰越大,可靠性越低。为此,越是长的总线,其线数就应尽可能减少。数据总线的宽度有一位、一个字节或一个全字等等。

在满足性能要求以及和所用通讯类型和速率相称的情况下,应尽量减少总线的线数。采用线的组合、串/并转换和编码技术可以实现这点,但通常会降低总线流量。

采用线的组合能减少只按功能和传送方向所需的线数。例如,性质相似、方向相反且不同时发生的二根单向线可用一根半双向线代替;又例如,可采用在少数几根多功能线进行编码来代替每种功能都单独用一根线完成的很多根单功能线等等。串/并转换是在总线二端经串/并、并/串转换器转换,以便使用较少的线数,经多次传送后再在端点转换成完整的字。正如前面说过的原因,这种串/并的程度是根据系统成本和性能的折衷平衡选取。极端的一位串行传送的总线只用于远距离通讯。

上面我们分析了总线的控制方式、通讯技术、数据宽度和总线线数等;可以看出,一旦确定了I/O总线的这些基本性能之后,总线的申请、使用方式以及相应的规范也就确定了,所有要使用总线的设备或部件都必需遵守这个规范,否则,即使是电气上能连到该总线上,也无法取得总线的使用权和传送信息,而且还有可能使整个总线的工作被破坏。因此,所谓I/O设备(确切讲是I/O控制器)的“接口”就还需包括能满足I/O总线要求的申请、使用规范。

既然这样,那I/O总线的标准化问题就应引起我们的重视。目前,各个厂家生产的不同型号机器的I/O总线规范都不同,这就使得I/O设备厂或是用户要将某个I/O设备接到某种型号机器时,往往需要专门设计其“接口”,从而不能使各种I/O设备能通用于不同型号的机器,这是不方便的。好在在同一个系列内,其I/O总线的规范是相同的,因此能实现各个I/O设备可通用于各档机器。不过,随着微处理器的出现,I/O总线的标准化问题开始得到解决。这是因为I/O设备厂以及微型机的用户很难为各种微型机配上不同的I/O接口。于是,在七十年代中期就出现了微型机用的所谓S-100总线,目前已在不少微型机以至小型机中得到应用。这样,凡是具有S-100总线接口的I/O设备就能接到所有采用S-100总线的各种微、小型机。LSI、VLSI的发展支持了I/O接口的标准化,例如,在七十年代末期就产生了另一种标准I/O总线,IEEE-488总线的LSI接口片子,进一步简化了I/O设备与IEEE-488总线的相连。

至于I/O总线的设计,从系统结构设计者看,主要是流量和接口的确定。I/O总线的所需流量取决于该总线所接外部设备的个数和种类,而总线的价格一般是正比于其流量,尤其当所需流量超过某个范围时,其价格将迅速上升;而且,I/O设备的接口环节价格也随之增加。因此,若所设计的流量值过大时,不如采用多个总线,并限制每个总线的长度及所允许加接的I/O设备数。另外,如果总线流量小于所接多台I/O设备的平均流量总和,则可能使I/O设备难以申请到对总线的使用权,致使它们的效率发挥不了。我们要注意一定不

能使 I/O 总线成为 I/O 系统的瓶颈。

§ 3 中 断 系 统

五十年代中期就提出了中断的概念。随着计算机系统的不断发展和功能的加强,中断系统不断完善,它已经成为不只是 I/O 系统,而且还是整个计算机系统必不可少的重要组成部分。虽然我们把中断系统放在本章来讲述,但它不只在输入、输出过程,而且在更多的方面也起着重要的作用。本节首先简述中断系统的基本概念,着重于中断的响应和处理过程以及中断系统的作用;然后结合 IBM370 叙述中断的分类和分级;最后讨论一下中断系统中的软、硬功能分配。

3.1 基本概念

所谓中断,我们指的是对现行程序的中断。当出现来自系统外部、机器内部乃至处理器本身的任何异常的或例外的事件(例如机器故障、程序出错、外设请求、实时请求等等)时,CPU 应能暂停执行现行程序,转去处理这些事件,待处理完后再返回来继续执行原先的程序。这些事件是通过发中断请求的方法告知计算机。中断系统使 CPU 能及时处理和记录运行过程中出现的各种中断请求。

引起中断的上述那些事件,我们统称它们为中断源。CPU 硬件停止执行现行程序,转入对中断请求的处理,称为对中断的响应。若同时有多个中断请求,而对其中某个或某几个中断请求,CPU 处于响应状态就称为对这些中断源开放,否则就称为对这些中断源屏蔽。对有多个中断请求,需要按其重要性和紧迫性进行优先级排队,以决定对哪个中断请求进行响应。在对该中断响应后,就进入处理该中断的中断处理程序。在执行此中断处理程序时,若又产生新的中断请求,且允许被响应的话,就转去执行处理新中断的中断处理程序,处理完后,再来执行原来的中断处理程序,这称为嵌套式的多重中断。

看出,由现行程序切换到中断处理程序是经过由硬件实现的中断响应。在中断响应过程中要改变某些寄存器、计数器的内容,还要访问主存;然而,中断响应硬件的操作不是,也不能由机器指令指挥,而是由中断请求启动。它对程序员是透明的。中断响应硬件设计的好坏,对于中断响应速度、中断处理程序的复杂性、是否能完好地恢复中断现场、准确地回复到原来的中断点(我们在上一章“流水方式”那节已提到过)等都有直接的影响。它是系统结构和操作系统的一个重要交界面。可以说,在中断过程中,“硬”的部分是“中断响应”,“软”的部分是“中断处理程序”;系统结构设计者应该和操作系统设计者密切配合,合理地分配这二部分的功能。

大家知道,在进入中断处理前,需要保存好中断现场;然而,只是现场中的关键硬件状态是经中断响应保存的,而现场中的其它硬件状态及软件状态都是由中断处理程序保存的。系统结构设计者要慎重确定哪些硬件状态需经中断响应保存,这点我们以后还要讲到。

中断响应的另一个主要任务就是要解决好中断处理程序的进入。为了能快速、简便地实现现场保存和进入中断处理程序,系统结构设计者逐步摸索到了交换程序状态字(IBM 360/307称之为 PSW)的好方法。为保存中断现场,现行程序的程序状态字必须有中断点的指令

地址，此外，还得有中断点的某些机器状态，如条件码、指明现行程序属“目态”还是“管态”的状态位、以及现行程序为控制上述中断屏蔽的屏蔽位（当然不是一位）等，有的机器还包含有第二章讲过的上、下界寄存器的内容和其它状态位。这个程序状态字称为旧程序状态字。为进入中断处理程序，它的程序状态字必须有入口地址（即中断处理程序的第一条指令地址）以及中断响应告诉中断处理程序有关该中断的状况（称为中断码），此外，也得有该中断处理程序所要用的中断屏蔽位及其它状态。这个程序状态字称为新程序状态字。

看出，只要完成新、旧程序状态字的交换，机器就能从现行程序切换到中断处理程序。新程序状态字是存在主存，而旧程序状态字也应保存在主存。然而，中断响应不是由机器指令指挥的；因此，新程序状态字的取出和旧程序状态字的存入都不能由“取(存) PSW”机器指令实现。也就是说，存放新、旧程序状态字的单元地址不能由机器指令地址码指明，而得由中断响应硬件自动形成；所以，这些单元的地址是固定的。有的机器是对所有中断都只有一个中断程序入口，而有的机器（如 IBM370）则是对各类中断分别有不同的入口；对于后者，需由中断响应硬件按不同的中断类别形成对应的新、旧程序状态字地址。显然，交换 PSW 的过程不能被中断；然而，在这个过程中可能出现“访主存错”，系统结构设计者应考虑到这点，采取相应的措施。

由上述关于现场保存的讲述可以看出，中断响应只能保存指定的这些硬件状态；为了能准确地恢复中断现场和返回到中断点时能从指令的起点开始执行，CPU 就不是在任何时候都能响应中断。如果安排在一条指令的执行已结束，下条指令的解释未开始时才能响应，那对现场保护和中断点的返回都是有好处的。然而，并不是总能做到这点，一种情况是如果指令（如字符行指令）的执行时间过长，使得在指令执行期间出现的中断请求可能等待过久，这会延误某些紧急情况的处理；所以有的机器对这些指令采用在执行过程中也允许中断的方式。另一种情况是由于出现了使现行指令执行不下去的中断，如“访主存错”和下一章要讲到的“页面失效”等，“逼”得机器在指令的解释或执行过程中进入中断。系统结构设计者要细微地分析所有这些可能性，正确地解决好现场保存和返回后从指令的哪一点继续解释或执行。

进入中断处理程序之后，先要继续保存现场，接着按中断源的要求进行处理，而后恢复现场，返回到中断点。这些都是要通过机器指令实现。中断处理程序的执行和其它程序的执行是一样的。中断处理程序应是可中断的，因为在中断处理过程中应能响应和处理优先级更高的中断；但是，为返回到中断点而进行的交换 PSW 过程不能被中断。中断处理程序应是可再入的，因为对嵌套式的多重中断，某个中断处理程序可能被多个中断请求（如多个外设）所共用。

从上述中断过程可以看出，中断与执行子程序相似，它需要记住返回地址，也可以象子程序那样套迭起来。但是，从主程序的哪一点上入子程序、怎样进入，是事先安排好的；而从哪一点中断机器正执行的程序以及何时转入执行中断处理子程序却往往不是事先可以安排的。注意，这里说的是“往往不是”，对于处理机内部的事件如上溢、下溢、非法地址、非法操作、被“零”除等程序性中断有时是有意事先安排的，至于管理程序调用中断更是如此。因此有时把这类中断称为陷阱以区别于其它意外的中断。陷阱能够再现，是直接由程序引起，而意外中断是无法再现的。所谓再现，就是说，如果程序重复运行，则每次这个陷阱将在同样地方再次出现。此外，子程序是用来提高主存利用率，简化程序设计和模块化的一种

局部性手段，而中断系统却关系到整个计算机系统的功能。

那么，中断系统在计算机系统中究竟起着什么作用呢？

首先是实现分时操作所必需的。对于 I/O 系统，中断系统可被用来作为低速的外部设备和高速的中央处理机联系的手段。通道和外设通过中断系统向中央处理机报告，例如申请交换数据、数据交换完、交换出错或某外设可恢复使用等。这样，多台外设通道程序控制下就能各自独立地与中央处理机并行工作，从而大大提高整个系统的工作效率。中断系统是用于变更正在执行的程序流程的有效手段。多道程序和分时操作的选择与换道，没有中断系统是不可想象的。

第二是监督现程序，提高系统处理事故的能力，增加系统的可靠性。例如，当处理机在运算中发生溢出、地址错、非法操作码等程序性错误或机器故障时，就可通过中断系统暂停现程序、保留现场，转入相应的中断处理程序加以适当处理或通过调出操作系统进行程序复执，排除偶然性故障；或是将错误和故障加以记录，以便为故障诊断和恢复作准备。又如，为了便于调试程序，中断系统还可以对现程序进行跟踪，例如，“指令地址符合跟踪”。通过跟踪，建立中断请求，使机器进入相应的中断处理程序，把这道程序挂起或打印出来供检查。

第三，它是实现实时处理必不可少的工具。由于实时信息是随机的，只有通过中断系统及时响应和处理，才能避免信息的丢失和错误的动作。

第四是作为人干预机器的手段。无论是用户通过终端设备如显示终端、控打、光笔等，还是操作和维修人员通过控制台外中断键干预，都必须经中断系统，避免因他们的错误动作而破坏机器的正常工作，影响机器的使用率。

第五是用于实现多机系统中各机间的联系。在多机系统中可互相通过中断系统发中断请求和响应来交换信息。在双工或多工系统中，某台机器出现告急情况可通过中断系统进行联系，实现同步和自动切换。

第六是用于实现目态程序和操作系统的联系。通过在目态程序中设置“访管（调用管理程序）”指令来中止执行现程序，在中断系统控制下转而执行某个管理程序。

总之，中断系统在计算机系统中具有很重要的作用。为了要完成以上的功能，从中断的过程来看，是要用到许多与中断有关的硬件和软件的。中断系统和操作系统是密切相关的。在很多方面，操作系统是借助中断系统来控制和管理计算机系统的。

3.2 中断的分类和分级

中断系统与指令系统一样，随计算机的功能和结构格式而异。譬如说，对用于实时控制的机器，要求中断系统处理随机信息的能力强一些；对用于科学计算的机器则要求处理程序性中断的能力要强一些。而对于科学计算，实时控制和数据处理兼用的通用计算机就要求其中断系统比较灵活、完善，并应为用户提供自己设计中断处理程序的可能。

为处理每一个中断请求，都必须调出相应的中断处理程序。如果中断源（包括各种 I/O 设备）比较少，通过中断系统硬件可以比较方便地对每个中断源形成入口，进入相应的中断处理程序。然而，对中、大型，多用途机器来说，中断源是相当多的，一般可达数十至数百个。若为每个中断源设置一个中断入口，那在硬件实现上是不利的，在中断处理上也是没有必要的。因此，就必须对中断源进行分类，把性质比较接近的中断源划为同一类，对于每一

类给定一个中断处理程序入口，再由软件负责对同类中的每个中断源分别予以处理。下面我们以 IBM 370 为例来说明这种分类。

IBM 370 把中断分成六类：机器校验、管理程序调用、程序性、外部、输入/输出和重新启动。前五类中断只发生在 CPU 处于运行状态时，而重新启动指令不论 CPU 是停止状态还是运行状态都可以发生。这六类中断，它们的旧 PSW 和新 PSW 所在的存贮单元位置都是各不相同的。每类的具体中断原因可由中断码进一步指明，或是由中断期间放在指定存贮单元中的附加信息指明。

机器校验中断是告诉程序发生了设备故障。用 64 位机器校验中断码指明故障原因和严重性，更为详细的中断原因和故障位置可由机器校验保存区内容提供。这里包含有电源故障、运算电路的误动作、主存出错、通道动作故障、处理器的各种硬件故障等等。

访管中断是发生在执行访管指令时，CPU 不能禁止该中断。

程序性中断是包括指令和数据的格式错、程序执行中出现的异常（非法操作、目态下使用管态指令、主存保护、寻址超主存容量、各种溢出、除数为 0、有效位为 0 等）以及程序的事件记录、监督程序对事件的检测引起的中断等。

外中断来自机器内部或外部，它包括各种定时器中断、外部信号中断及中断键中断。各种定时器中断用以计时、计费、控制等用。外部信号中断主要用于与其它机器和系统的联系。中断键则用于操作员对机器的干预。这些外中断又可再分成二类。一类是若未被响应继续保留，另一类是如不响应则不再保留。

输入/输出中断是 CPU 与 I/O 设备和通道联系的工具。

管理程序调用、程序性、外部、I/O 这四类中断的中断码均为 16 位。

重新启动中断是为操作员要启动一个程序所用。CPU 不能禁止这种中断。其中断码为 16 位全 0，或者不提供，取决于控制方式是基本的还是扩展的。

以上以 IBM370 为例说明了为什么把中断分类以及一般可以分成哪几类。现在来讲述中断的优先级问题。

由于中断源很多，它们相互独立而随机地发出中断请求，因此常常会同时发生多个中断请求。对于同一类型中的各中断请求，其响应和处理仍有其优先次序，但通常不由中断系统的硬件而是由其软件或通道来管理。而对于不同类型的中断，就要根据中断的性质、紧迫性、重要性以及软件处理的方便性把它们分为几级。中断系统按中断源的级别高低来响应。通常把优先级最高的中断定为一级，其次是二级，再次是三级……。优先级高、低的划分不同机器有其各自的特点。一般中、大型机器把中断响应的优先级按排成机器校验为第一级；程序性和管理程序调用为第二级；外部为第三级；输入输出为第四级；……。我国的 DJS-200 系列机也如此划分，它把五类中断分成四级。

把机器校验放在第一级是因为象掉电、CPU 的地址、数据以及通路错等会使机器无法正常工作，必须及时处理，否则将影响整个系统的正常运转。然而，对于只影响到局部的某些故障，优先级可以低一些，例如 200 系列中将某个通道或外设的故障常常放在输入输出这一级。IBM370 把机器校验分成紧急的和可抑制的二种，分别属不同的优先级。

程序性中断和管理程序调用通常列为第二级，因为若让程序性中断级别低于外部中断和输入/输出中断，那么在同时出现这三类中断时就会先响应外部中断和输入/输出中断，而如果在处理这些中断的管理程序中又出现新的程序性错误时，产生的程序性中断就可能与原先

的程序性中断源混在一起。因此，应优先响应程序性中断，然后再响应外中断和输入/输出中断。这样，如果在处理外部中断或输入/输出中断的管理程序中出现新的程序中断时，不会导致混乱。

外中断级别高于输入/输出。因为它涉及到多机联系、人机干预控制操作等，而输入/输出中断的中断源一般只是某台外设的请求，相比之下后者属于局部性问题，而且它们还可由各通道管理，中断响应晚些也不致丢失信息和带来太大的影响。

重新启动中断级别一般最低，其理由是很明显的，因为重新启动的时间并不是那么紧迫的，然而当CPU处于停止状态时，重新启动就应具有比挂起的输入/输出，外部或可抑制的机器校验中断都要高的优先级。

至于管理程序调用中断，与其它几种中断在性质上是有差别的。管理程序调用中断是在现行程序中安排一条“访管”指令中断机器正在执行的程序，通过中断系统调出相应的管理程序。因此，它不是因为故障和错误强迫机器进入管理程序的，而是自愿地进入。所以，有时把它称为自愿性中断；而其它类别的中断，何时中断现行程序往往不是事先能安排的，是出现请求后强迫CPU进入管理程序来进行中断处理，所以有时把它们称为强迫性中断。这种管理程序调用的自愿访管中断按排在第二级是因为在机器执行这条“访管”指令时，如果发生紧迫的机器故障和错误，只有先去处理故障和错误，在机器没有故障和错误后方能根据“访管”指令的功能，使机器进入管理程序。由于其中断响应不是靠中断级的排队，而是靠机器执行“访管”指令，所以任一级的中断屏蔽都应不能屏蔽管理程序调用中断，即它不受中断级屏蔽的控制。这就可以使各级中断的管理程序都可使用“访管”指令，嵌套进入相应的管理程序，从而给系统程序的编制带来方便。

在有的计算机系统中，往往还有0级中断。这是在机器因故障重迭发生或无法排除，完全不能正常工作，而由中断系统硬设备发出的机器告急。它或者向操作人员报警请求直接干预，或者向它机发出求援，进行机器间的任务切换。这种告急状态也是机器所执行的程序中断，但它并不是真正的中断级，它并不是参加中断级排队，中断管理程序也没有它的入口，被中断后一般也无法自行恢复。

IBM370中断响应的优先次序为：紧急的机器校验；管理程序调用；程序性；可抑制的机器校验；外部；输入/输出；重新启动。

以上讲的是在同时发生多个中断请求时，中断响应的优先次序，即由中断响应硬件中的排队器所决定的响应次序。然而，前面讲过，中断的处理是由中断处理程序实现的，而中断处理程序是可以在具体处理前被中断的；所以，中断处理次序就可以不同于中断的响应次序。这是软、硬结合的好处，下面我们就来讲述。

一般情况下，在处理某级中的某个中断请求时，与它同级的或比它低级的中断请求应不能中断它的处理，而应是在处理完该中断返回到主程序后，再去响应和处理这些中断。但比它高级的中断请求却应该能中断它的处理，即CPU应转去处理比它更高级的中断请求，而后再返回来处理原先的这个中断请求。图4.15示出了这个过程的一个例子。

看出，对此例，在执行第2级中断处理程序时，应能响应第1级中断请求，但却不能响应第4级中断请求。设各级中断屏蔽位为“1”对应于开放，为“0”对应于禁止（屏蔽），为实现上述中断处理次序与中断响应次序相同，各级中断处理程序的中断屏蔽位应安排如下：

中断处理 程序级别	中断级屏蔽位			
	1 级	2 级	3 级	4 级
第 1 级	0	0	0	0
第 2 级	1	0	0	0
第 3 级	1	1	0	0
第 4 级	1	1	1	0

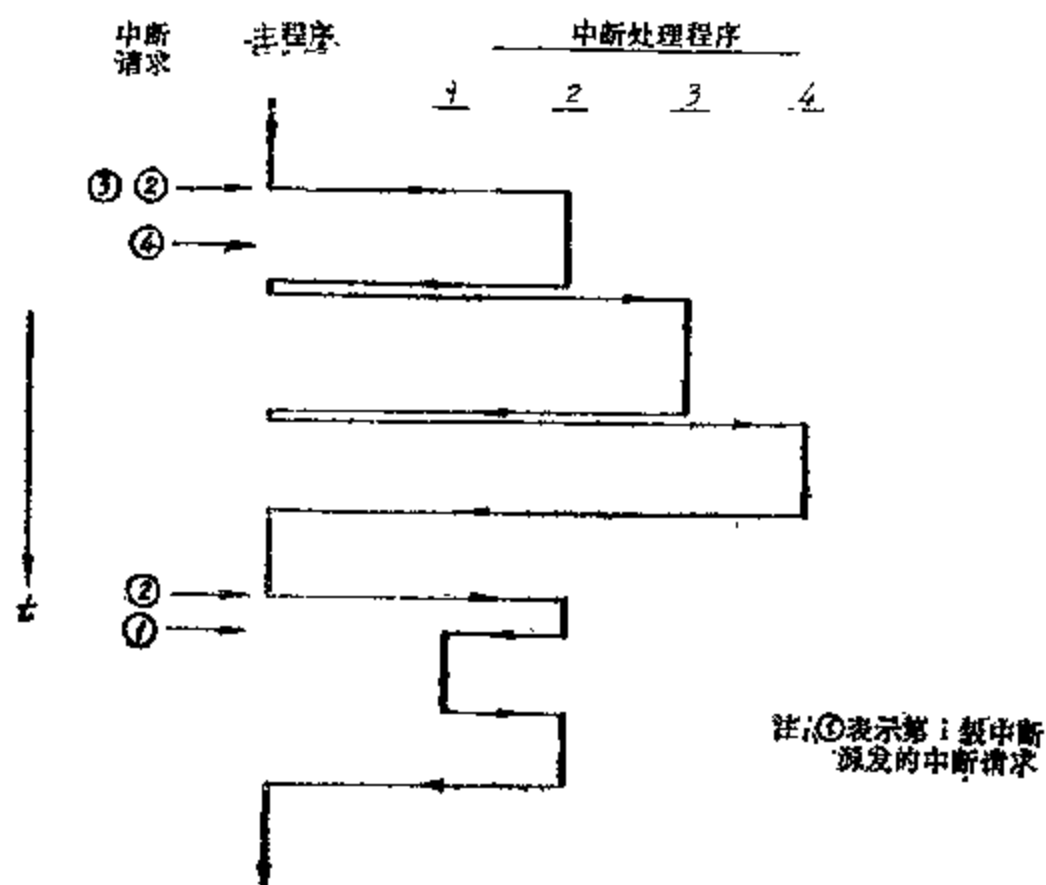


图 4.15 多级中断的处理示意图

循此，改变中断级屏蔽位的状态，我们就能使中断处理的次序不同于不能改变的中断响应次序。例如，若要使中断处理次序由 1→2→3→4 改为 1→4→3→2，则只需使中断级屏蔽位改为如下所示：

中断处理 程序级别	中断级屏蔽位			
	1 级	2 级	3 级	4 级
第 1 级	0	0	0	0
第 2 级	1	0	1	1
第 3 级	1	0	0	1
第 4 级	1	0	0	0

这种情况下，当所有 4 级中断请求同时出现时，其处理过程就如图 4.16 所示。中断响应硬件是只能按照由 1 级到 4 级的次序响应，所以第 1 级中断先被响应，并先处理；接着响应第二级中断，但一进入第 2 级中断处理程序，就因为它的中断屏蔽位对第 3、4 级中断是开放的，所以就经中断响应进入第 3 级中断处理程序；由于第 3 级中断处理程序对第 4 级中

断也是开放的，所以又被中断而进入第 4 级中断处理程序，从而实现了先处理第 4 级的要求；而后，返回到原中断点，即第 3 级中断处理程序，处理完第 3 级中断后才返回到第 2 级中断处理程序，去处理第 2 级中断。

由于各级中断处理程序的新程序状态字是存在主存中的指定单元，所以就能由操作系统根据需要改变各级中断的实际处理顺序。

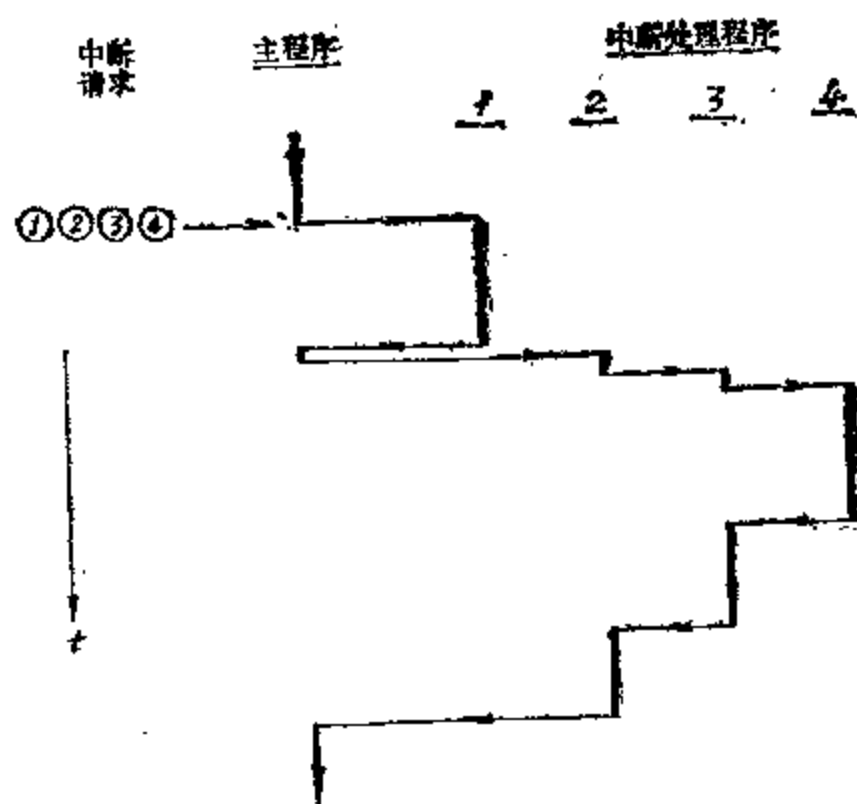


图 4.16 1→4→3→2 的处理顺序示意图

3.3 中断系统的软硬功能分配

从前面的分析可知，中断的全过程无非是包括优先级的确定，中断现场的保存，对中断请求的分析和处理以及返回中断点等，而这个全过程是由中断响应硬件和中断处理程序共同实现的。因此，中断系统的软、硬功能分配实质上就是中断响应硬件和中断处理程序的功能分配。它对中断系统性能的好坏影响很大，系统结构设计者和操作系统设计者都应仔细、全面地分析。

最初的中断系统，它的中断响应硬件很简单，从优先次序的排定到确定是由哪个 I/O 设备发的中断请求，全由软件用程序查询的方法实现。这符合当时使硬件尽量简单，而把更多的功能交给软件去实现的思路。这样一种中断系统，它的中断响应时间和中断处理时间都必然过长。后来，随着中断使用经验的积累以及硬件价格的日益下降，就逐步扩大了中断响应硬件的功能。

以优先次序的排队来说，很快地就尽可能转为由硬件实现。从类似于上节确定总线使用权的“串行链”方法（即优先次序是由各个 I/O 设备离 CPU 的远、近来定）发展成前述把各个中断请求按类分级，而由中断响应硬件的排队器按固定次序选定的方法。

以中断源的确定来说，很快就由程序查询改进为由设备编码形成（直接或经入口表）各个设备的中断处理程序的入口地址；进而发展成为上述的对每级中断经中断响应硬件形成该级中断程序状态字地址的入口方法，并把中断源的状况以中断码的方式，经旧程序状态字告知中断处理程序。

至于中断现场的保存，前面说过，现场包括软件状态（如作业名称、级别，上、下界值，各种软件状态和标志等）和硬件状态（如现行指令地址，条件码等状态信息，各种控制寄存器内容以及能被所有程序使用的通用寄存器内容）。软件状态宜于经中断处理程序保存，一是它们本来就是存在主存内，二是它们的数量可能随着操作系统的发展而扩大。而硬件状态的保存，是全经中断响应硬件保存呢？还是部分经它，部分经中断处理程序保存呢？至今各个机器并不一致。由于有些硬件状态是一般机器指令访问不到的，若为各种硬件状态

都设置专用指令，那会使这部分指令膨大到很不合理的地步，因此，这些硬件状态应经中断响应硬件保存。这点很快取得一致，并形成了把这些本来分散于 CPU 各个部分的硬件状态综合成“程序状态字”的思路。然而，对于程序状态字的位数究竟多大合适，各个机器有不同的思路，主要是通用寄存器和变址寄存器的内容是包括在程序状态字内，还是由中断处理程序保存，这点我们在下面还要讲到。

那么，是否由中断硬件实现上述中断全过程的操作愈多就愈好呢？不是的。对于中断系统的软、硬功能分配，要考虑到二点：一是中断响应速度，二是中断系统所应具有 的灵活性。

直至进入中断处理程序的中断响应时间是中断系统的一个重要性能指标，对于实时系统更是如此，这是因为它的中断处理可能比较简单，但却要求能及时、快速地响应。从前面的分析可知，对所有各种中断，不论其中断处理时间多长，其中断响应时间都是一样的，主要取决于交换程序状态字的时间。对于 IBM370，程序状态字为 64 位，等于它的长字，所以为交换程序状态字至少需二次访主存，读和写长字。显然，要经中断响应硬件保存的硬件状态愈多，程序状态字就愈长，所需的访存时间就愈长，响应速度就愈慢。尤其是随着通用寄存器数目的增加（这是当前的发展趋势，极端的例子是 CRAY-I，它需保存的向量、标量、地址等寄存器的总位数达 39000 位），要使通用寄存器内容的保存经中断响应硬件实现，势必会使中断响应速度慢到不能用的地步。

我们要知道，并不是所有中断处理都需要把通用寄存器的内容，或是整个通用寄存器的内容都保存起来。这是因为经中断进行的任务切换可以有二种，一种是整个任务的切换（如由某道程序切换到另一道），这当然需要把整个通用寄存器的内容都保存起来；另一种是某道程序调用某个管理程序的任务切换，对于具有通道的机器，由于 CPU 的 I/O 管理程序动作简单，从而在切换时可能不需要保存（或只需保存部分）通用寄存器的内容。因此，目前一般的机器，各种通用寄存器的内容是由中断处理程序按切换需要保存。

由于任务切换中的软、硬状态保存时间是影响操作系统性能及其设计的一个重要参数，所以系统结构应为通用寄存器内容的保存提供更好的支持。一种方法是设置“通用寄存器转贮”指令，使得可由一条指令实现通用寄存器内容的保存，以减少为保存通用寄存器内容那段程序所耗费的取指令时间。另一种方法是在机器内设置多套通用寄存器组，规定每个任务只能使用其中的一套；这样，在任务切换时就可做到不必切换、保存通用寄存器的内容。然而，由于通用寄存器的套数不能多（有的机器取为 4 套），这种方法比较适合于任务数较少但要求快速处理的实时系统。对于采用主存多体交叉（下章要详细讲）的巨、大型机，可在一个主存周期内读、写 m 个字，那增长程序状态字达 m 字长比单字长的程序状态字，其中断响应时间差别不大；那就可使通用寄存器（至少是其部分）内容组合在程序状态字内。

我们在前面讲的都是立足于集中处理的计算机系统（即所有任务都在同一个处理机内实现）。然而，对于如上一章图 3.29 所示的那种局部分布处理系统，就有可能显著减少每个处理机内任务的切换次数，有可能做到对于 I/O 系统的输入、输出不需要中断中央处理机的用户程序。随着微处理器的发展，从根本上简化任务切换的分布处理思路会被更多的人所接受。HP-3000 就已采用了这种思路，我们在“外围处理机”那节再来讲述。

中断系统必须具有灵活性，因为它所面对的是种类繁多，要求殊异的中断源，因此，必需是软、硬结合。前面讲过的，通过软、硬结合使得能灵活地改变各类中断的处理次序就是

明显的例子。

总之，中断系统的软、硬功能分配是系统结构设计者需要认真考虑的一个方面，我们在这里只是讲了一些情况和思路。

§ 4 通 道

本节主要以 IBM370 为例，先叙述通道方式 I/O 系统的基本概念，如通道的分类及其功能、通道传送数据的方式、通道流量的分析计算、接口与总线等。接着叙述有关的输入输出指令、通道指令、通道状态字和输入输出中断，并以一个例子介绍通道方式的输入输出过程。

4.1 基本概念

4.1-1 通道的分类

采用通道方式组织输入输出操作，通道从 CPU 和主存获得控制信息，代替中央处理机来组织外部设备与主存间的信息交换。这样，输入输出系统采用了如图 1.15 所示的“中央处理机/主存—通道—设备控制器—外部设备”这样的四级连接方式。除了中央处理机能运行多道程序外，中央处理机能与通道并行操作，各个通道之间、同一通道的各个外部设备之间也能并行操作。同时，也给用户增接外部设备提供灵活性。

我们在 §1 讲过，通道是 I/O 处理机，它应该是独立于 CPU 的处理机实体。然而，采用通道结构的 IBM360 是系列机，它的高、低档机器的价格差别达 200 倍以上。对于低档的机器，如果是设置专用的 I/O 处理机，那其价格就便宜不了；但是，在第一章 §3.1 我们讲过，IBM360 既然是系列机，那它的所有档，都应该具有相同的从程序员看的系统结构。所以，IBM360 的低档就使其 I/O 处理机（通道）尽可能与 CPU 共用硬件以降低造价；但这样一来，I/O 处理机只能与 CPU 分时使用这些硬件，即它们是“逻辑上独立，但物理上结合”。从而，从实体上，这种低档机器并没有独立的 I/O 处理机，但在概念性结构上，它仍然有通道结构，但它的通道是不能如高、中档机器那样和 CPU 并行工作。我们把具有 I/O 处理机实体的通道称为“独立型通道”，而把与 CPU 分时使用硬件的通道称为“结合型通道”，但从通道的概念结构来讲，这二者是一样的。

按输入输出信息的传送方式来分，通道可分为字节多路通道、选择通道和数组多路通道三种。这种划分与输入输出设备的传送速率有很大关系。我们一般把传送速率在 20K 字节/秒以下的外部设备称为低速设备；速率在 20K~200K 字节/秒范围内称为中速设备；速率大于 400K 字节/秒的称为高速设备。

一、字节多路通道

我们以光电输入机这样的低速设备为例，它的传送速率可达 1K 字节/秒，即传送一个字节需 1 毫秒，而通道处理一个字节只要几个微秒就足够了。为了等待光电机机械运动到下一排孔，通道要浪费 90% 以上的时间。显然，如果设计的通道能为多个低速设备以字节交叉的方式服务，就会使效率大大提高，字节多路通道就是这样提出来的。

字节多路通道结构表示在图 4.17，它有多个按字节方式传送信息的子通道，它们能独

立地执行通道指令。字节多路通道一般工作于“字节多路”方式，对应这种方式，它能同时为多个设备服务，以字节宽度传送信息，即通道与某个子通道交换完一个字节后，就转去为别的子通道服务。各个子通道间“并行”操作，以字节宽度“分时”进出通道。连接在每个子通道的多台设备也能“分时”使用子通道。显然，这种方式适合于连接大量的低速设备，因为低速设备的每个字符（字节）间有很长的等待时间。通道的总流量应大于该通道加接设

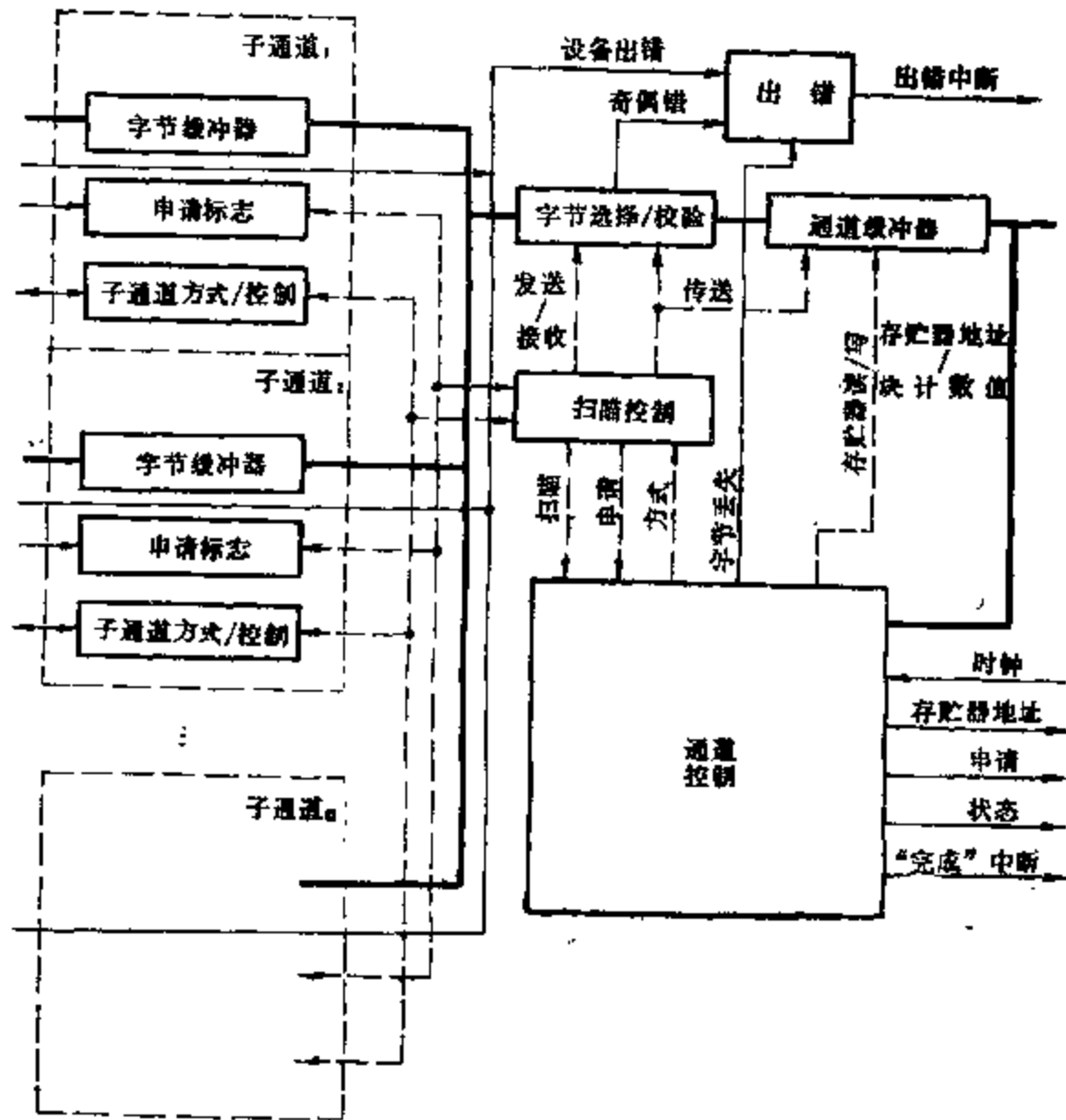


图 4.17 字节多路通道结构

备速率的总和。例如，如果通道的工作周期为 2 微秒，则该通道至多可以连接 500 台速率为 1K 的低速设备。

我们在 § 1 讲了在使用某个 I/O 设备时，需能递增该设备在主存 I/O 缓冲区的地址，递减信息块长度计数器，并判其结束否。对字节多路通道，每个设备所用的主存缓冲区地址不能（因为设备过多）也没必要（因为设备是低速的）存在通道内，而应存在主存内。

对于字节多路方式，就输出而言，其操作内容为：

- (1) 从主存读出所要传送的字节位置地址，然后加“1”并写回；
- (2) 按 (1) 读出的地址读出一个字节并把它传送到通道缓冲器；
- (3) 把字节再传送到子通道字节缓冲器，并访主存取信息块计数器值减去“1”、测试“0”并写回；
- (4) 查询别的子通道是否需要服务。

如果字节多路通道所接 I/O 设备的速度提高了，IBM370 也可把“字节多路”方式改

为使通道在一段时间内，固定地只为某个子通道服务。然而，由于这种通道的数据宽度（参看本章 § 2.4）为单字节，而且，为传送一个字节就需五次访主存：取、存该字节地址，取、存信息块长度值，访问该字节单元。所以，这种方式的效率很低，尽可能不去采用，且至多只能接中速 I/O 设备。对于高速 I/O 设备，应连接于下述选择通道。

二、选择通道

以磁鼓、磁盘这样的高速设备为例，它的传送速率可达 400K~1.5M 字节/秒。这样，字节之间的空余时间已没有多少可利用的，所以在数据传送期间，通道只能为某一高速设备服务，为此设置了选择通道。

选择通道每次只能从所连接的设备中选择一台，以进行数据的传送操作，亦即只能执行一道通道程序。数据传送是以成组（块）的方式进行的，每组织一次传送，就传送一个数据块，所以传输速率高。由于是成组传送，所以在传送前必须把主存缓冲区的首地址、数据块的长度（即字数）和设备号送入通道。

选择通道在功能上能够实现：

- (1) 每当与主存交换一个字后，能自动修改在通道内的主存缓冲区地址；
- (2) 同时修改在通道内的数据块长度值；
- (3) 为匹配外部设备和内部数据总线宽度，能装配或拆卸数据元；
- (4) 能进行单个字节的奇偶校验；
- (5) 能报告通道和设备状态和发中断；

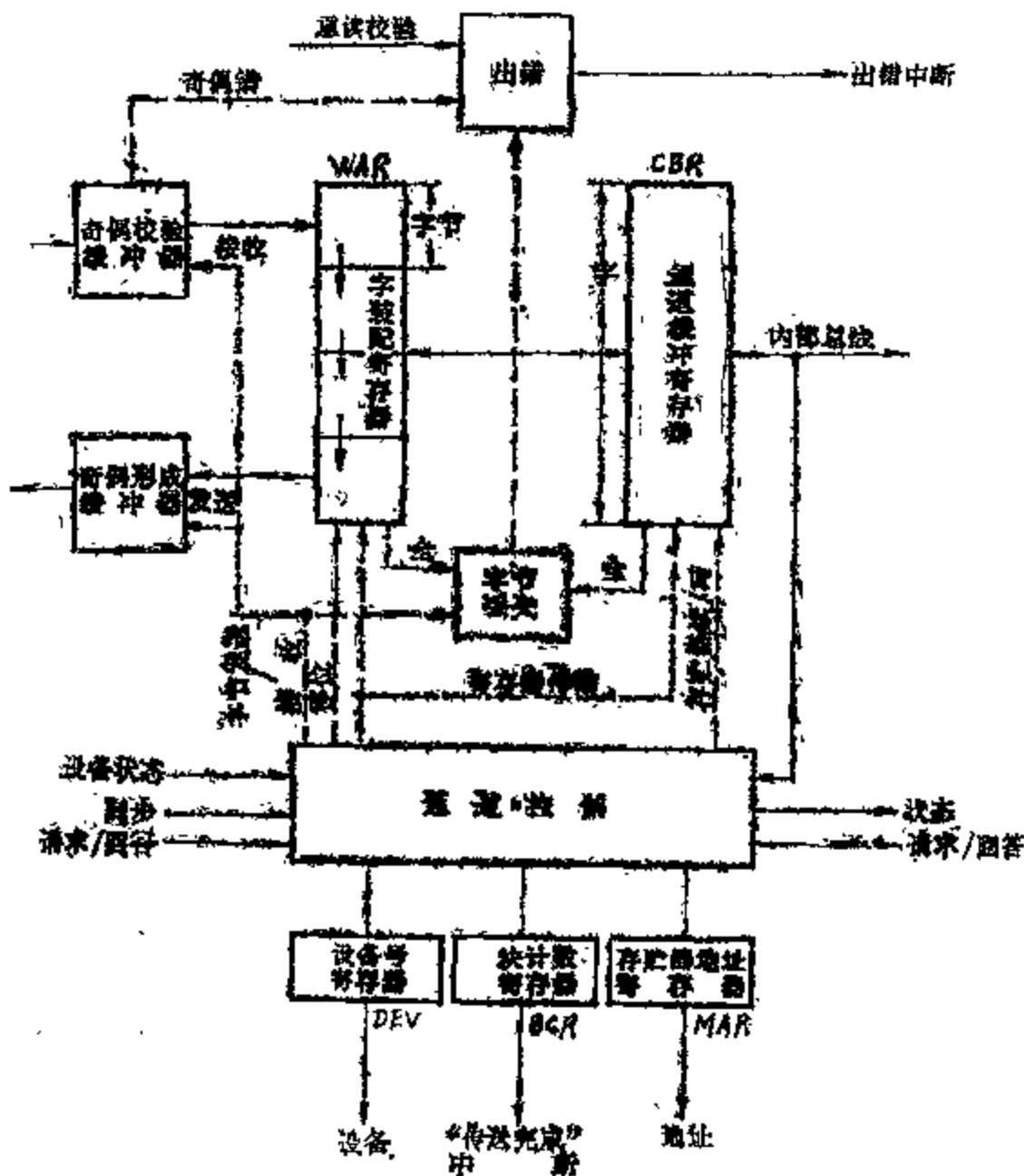


图 4.18 选择通道结构

(6) 能控制数据的正确传送。

图 4.18 表示了典型的选择通道结构。

图 4.18 中, 字装配寄存器(WAR)寄存要被传送到外部设备去或从外部设备接收来的字。发送到设备或从设备接收来的数据是按“字节”进行, 而通道与主存之间信息传送却是按“字”(IBM370 每字为四个字节)进行, 数据宽度的匹配是通过 WAR 的移位实现, 移位信号由设备发出。对输入而言, 待 WAR 右移填满后, 以并行的方式送往 CBR(通道缓冲寄存器), 再从那里由 CPU 时标同步传送至主存。对输出而言, 过程恰好相反。WAR 和 CBR 之间的传送是并行、异步的, 它们不能合并, 因为在当前的“字”被送到主存之前, 下一个字节可能到达。

现行主存地址存放在存储器地址寄存器(MAR)中, 在块计数寄存器(BCR)中保存未被传送的字数。每传送一个字后, MAR 和 BCR 分别加“1”和减“1”。当块计数寄存器的内容为“0”时, 产生“传送完成”中断, 告诉中央处理机本次传送已完成。如果在传送过程中产生奇偶出错或输入字节丢失时, 均产生出错中断。

三、数组多路通道

我们仍以磁盘、磁鼓这样的高速设备为例, 虽然它的传送速率很高, 但是在传送开始前, 需要进行寻址辅助操作, 这很费时间, 若采用上述选择通道, 那在这段辅助操作时间内, 通道在空闲。因此, 应尽可能重迭各台高速设备的辅助操作时间, 提高整个输入输出系统的流量, 这就是设置数组多路通道的原因。

数组多路通道集中了选择通道和字节多路通道的特点, 它有多个子通道, 可以同时执行多道通道程序。象选择通道一样, 存储器地址和块(字数)计数寄存器是放在通道硬件中, 它的加、减“1”动作能与数据传送同时进行。但它又象字节多路通道那样, 所有子通道能分时共享输入输出通路, 但它是成组交叉传送。这样, 它既具有多路并行操作的能力, 又具有很高的数据传送速率。

在执行通道程序时, 如果当前执行的通道指令是辅助操作而不是数据传送操作, 则通道将命令传送给设备控制器后就把该通道程序挂起, 但保存断点和必要的参数。控制器接收到此类辅助操作的命令后控制设备进行寻址。此时, 通道已可为别的通道程序服务。一旦外设控制器已结束寻址, 则控制器重新请求设备连接, 如通道能响应这个请求, 则数据按成组方式传送到主存, 在传送过程中, 整个通道为这道通道程序服务, 以保证高的数据传送速率。若控制器结束寻址时通道处于“忙”状态, 则设备和控制器保留其申请信号, 在所需信息块再次转到磁头下时重新发出。

当然, 这种通道最费设备, 往往还需在通道内设置高速小容量存储器。

对于上述三种通道传送数据方式, 可用下面的例子说明。若要发送 A、B、C 三个数据块信息, 它们分别来自三个设备, 每一数据块由如下的字节序列组成:

A: $A_1 A_2 A_3 A_4 \cdots A_n$

B: $B_1 B_2 B_3 B_4 \cdots B_n$

C: $C_1 C_2 C_3 C_4 \cdots C_n$

对于字节多路通道的字节交叉方式, 图 4.19 是可能的一种时间序列。

对于选择通道的传送方式, 下面的时间序列是可能的一种形式:

$A_1 A_2 A_3 \cdots A_n B_1 B_2 B_3 \cdots B_n C_1 C_2 C_3 \cdots C_n$

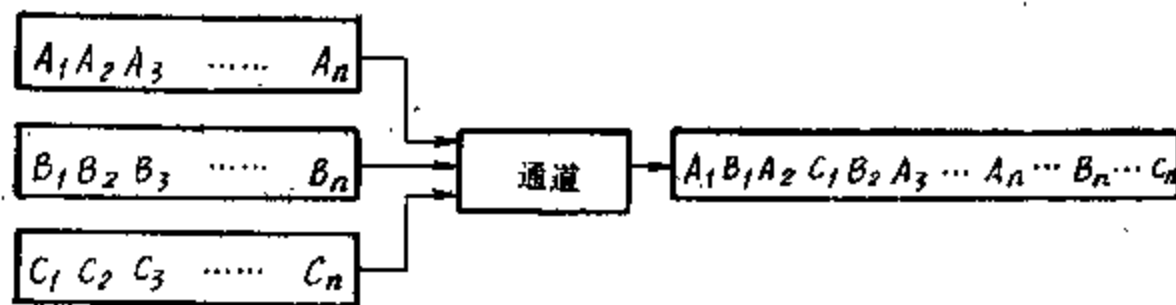


图 4.19 字节交叉传送方式

对于数组多路通道的成组传送方式，下面是可能的时间序列：

$$A_1 A_2 \dots A_k B_1 B_2 \dots B_k C_1 C_2 \dots C_k A_{k+1} \dots A_n B_{k+1} \dots B_n C_{k+1} \dots C_n$$

这里只是可能出现的传送序列，实际序列则取决于各设备的相对速度和在发送过程中各个设备的优先级。

通道的数据传送方式对整个 I/O 系统的性能有明显的影晌。D. T. Brown 用模拟方式对此进行了研究，其结果如图 4.20 所示。

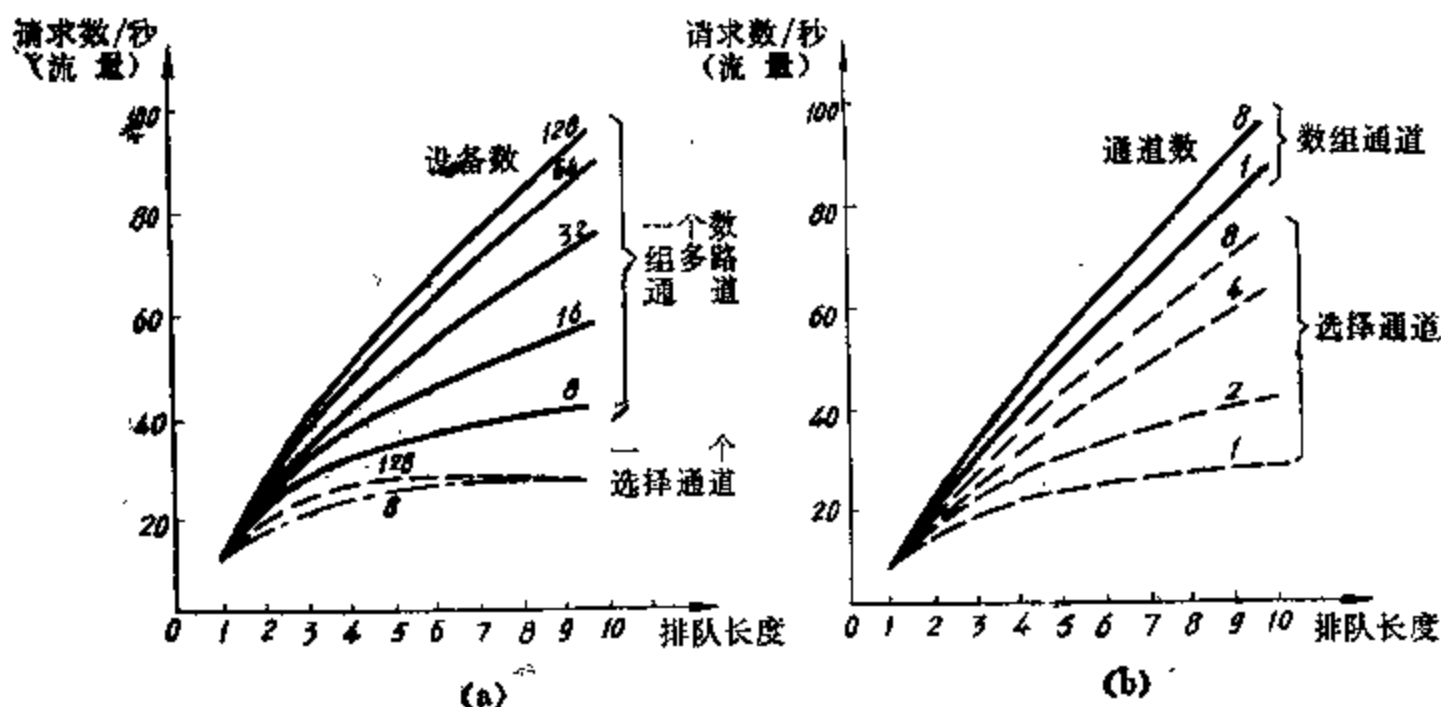


图 4.20 通道数据传送方式的模拟结果

图 4.20(a) 曲线表明在单个选择通道上，所接磁盘从 8 台增加到 128 台时，系统的流量并无明显增加，显然选择通道成了限制 I/O 系统性能提高的瓶颈；而当把选择通道改为数组多路通道时，系统的流量就随设备台数的增加而增加。图 4.20(b) 表示当把多台磁盘均匀分布于各子通道时，系统流量随着选择通道个数的增加而增加，但是数组多路通道个数的增加却并不显著增加系统的流量，可见数组多路通道存在着较大的潜力，每个通道还可加接更多台磁盘。看出，数组多路通道比选择通道优越。

4.1-2 通道流量的分析

上面我们介绍了三种类型的通道以及它们传送信息的特点，图 4.20 实验曲线表明通道的数据传送方式对 I/O 系统流量的影响。

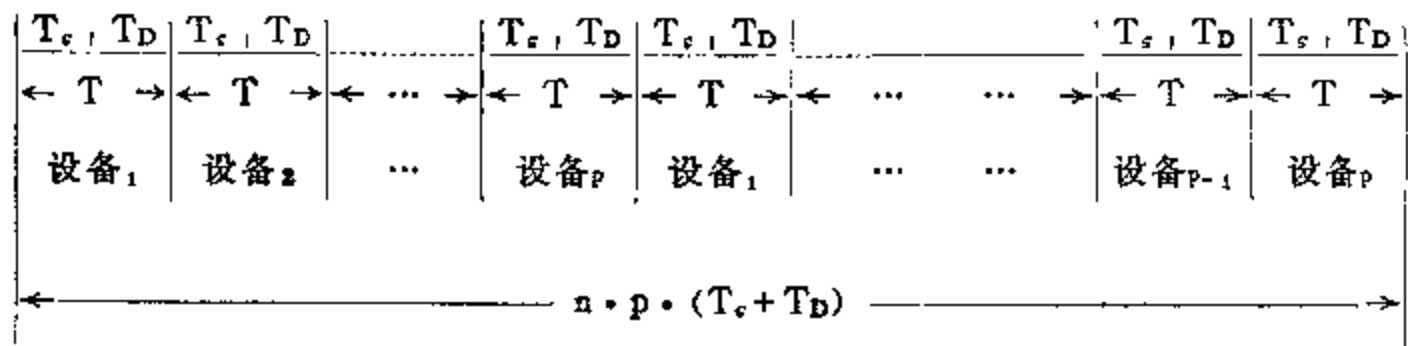
所谓流量，是指通道在数据传送期间，单位时间内传送的字节数。

通道与设备间的数据传送（详细过程在 § 4.2-5 再讲述），可分为三个阶段，即通道开始选择设备期、通道数据传送期和数据传送结束期。在通道开始选择设备期，通道根据“启

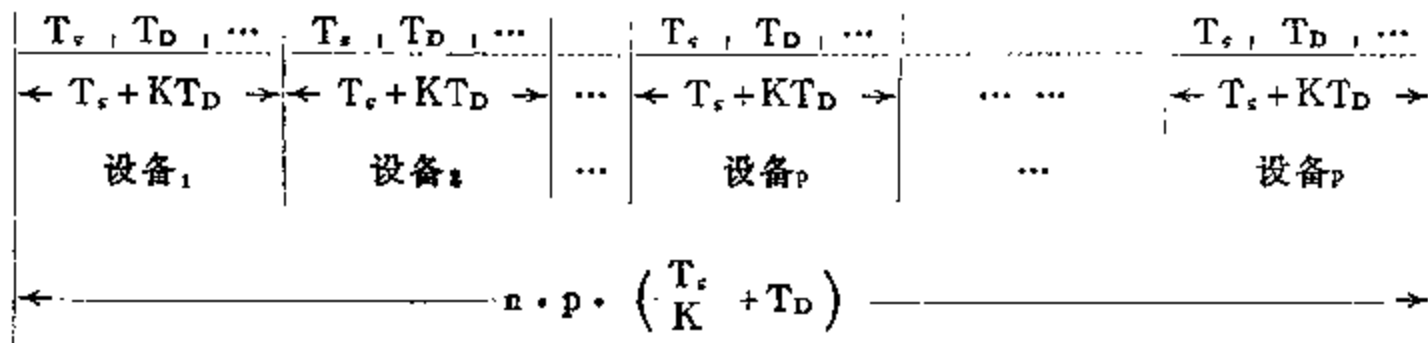
由 I/O 指令”所指明的设备地址去寻找对应的设备，并发有关命令。在通道数据传送期，当指定的输入输出设备被启动后，通道根据各设备的优先权响应请求中断的设备，实现设备与通道间的数据传送。此传送期还包括设备选择期。因为在通道上架接的多台设备同时工作时，还需选择当前要传送数据的是哪一台设备。数据传送结束期是指通道结束一次数据传送的过程。

我们感兴趣的是通道极限流量，因为它是设计通道的依据。所谓通道极限流量或通道最大传送速率是指通道在数据传送期间，单位时间内允许传送的最大字节数。

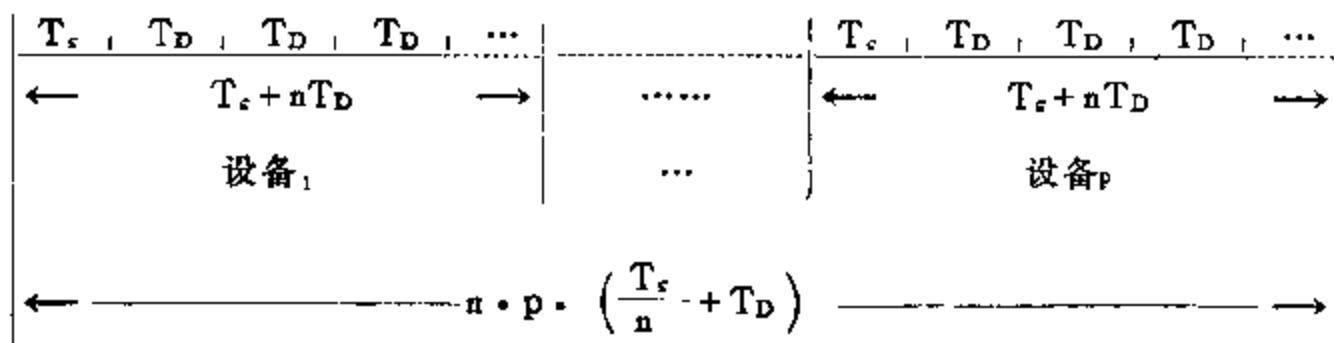
若每个通道接有 P 台设备，设通道数据传送期内设备选择需 T_s 时间，传送一个字节需 T_D 时间，则对应于上述三种通道，每台设备传送 n 个字节的过程如图 4.21 所示。由图可见，它们之间的区别在于通道的设备转接时间。在字节多路通道中， T_s 和 T_D 是交替的；在数组多路通道方式中，每传送一组 (k 个字节) 数据后，通道再选择设备，即每隔 k 个 T_D 才插入一个 T_s ；在选择通道中，则是一次传送数据结束后，才插入一个设备选择期，即每隔 n 个 T_D ，才插入一个 T_s 。



(a) 字节多路通道



(b) 数组多路通道



(c) 选择通道

图 4.21 一次数据传送过程

根据通道极限流量或最大传输速率定义，则有：

对字节多路通道

$$f_{\max \cdot \text{byte}} = \frac{1}{T_s + T_D} \quad (4.4-1)$$

对数组多路通道

$$f_{\max, \text{block}} = \frac{1}{\frac{T_s}{K} + T_D} \quad (4.4-2)$$

对选择通道

$$f_{\max, \text{select}} = \frac{1}{\frac{T_s}{n} + T_D} \quad (4.4-3)$$

由通道工作原理，通道的实际最大流量，对字节多路通道，为该通道所接设备的字节传送速率之和，而对于其它两种类型的通道应为所接设备字节传送速率中最大的一个。即

$$f_{\text{byte}, j} = \sum_{i=1}^{P_j} f_{i, j} \quad (4.4-4)$$

$$f_{\text{block}, j} = \text{MAX}_{i=1}^{P_j} f_{i, j} \quad (4.4-5)$$

$$f_{\text{select}, j} = \text{MAX}_{i=1}^{P_j} f_{i, j} \quad (4.4-6)$$

式中， j 为通道编号， $f_{i, j}$ 为第 j 通道中第 i 台设备的字节传送速率， P_j 为第 j 通道中所接设备的台数。

为了保证第 j 通道能满足所架接设备的流量要求，对于上述三种通道，应满足下述关系式

$$\begin{aligned} f_{\text{byte}, j} &\leq f_{\max, \text{byte}} \\ f_{\text{block}, j} &\leq f_{\max, \text{block}} \\ f_{\text{select}, j} &\leq f_{\max, \text{select}} \end{aligned}$$

如果 I/O 系统有 m 个通道，其中 1 至 m_1 为字节多路通道， $m_1 + 1$ 至 m_2 为数组多路通道， $m_2 + 1$ 至 m 为选择通道，则 I/O 系统的最大流量应满足

$$\begin{aligned} f_{\max} &= \sum_{j=1}^{m_1} f_{\max, \text{byte}} + \sum_{j=m_1+1}^{m_2} f_{\max, \text{block}} + \sum_{j=m_2+1}^m f_{\max, \text{select}} \\ &\geq \sum_{j=1}^{m_1} \sum_{i=1}^{P_j} f_{i, j} + \sum_{j=m_1+1}^{m_2} \text{MAX}_{i=1}^{P_j} f_{i, j} + \sum_{j=m_2+1}^m \text{MAX}_{i=1}^{P_j} f_{i, j} \end{aligned} \quad (4.4-7)$$

f_{\max} 是 I/O 系统对主存频宽 B_m (详见下一章) 的要求。大家知道，不仅 I/O 系统，而且还有 CPU 也需使用主存，在 B_m 中，I/O 系统占多大的比例是与机器的用途有很大的关系。各个机器对此有不同的考虑。有的认为，在 B_m 中可能有一半以上是被 I/O 系统占用。

至于 I/O 系统本身，可用 f_{\max} 与 $\left(\sum_{j=1}^{m_1} \sum_{i=1}^{P_j} f_{i, j} + \sum_{j=m_1+1}^{m_2} \text{MAX}_{i=1}^{P_j} f_{i, j} + \sum_{j=m_2+1}^m \text{MAX}_{i=1}^{P_j} f_{i, j} \right)$ 之差来衡量 I/O 系统的 (流量) 利用率高低。

下面举一个流量计算的例子。设有一字节多路通道，它有三个子通道，“0”号、“1”

号磁带机各占一个子通道；“0”号宽行打印机、“1”号宽行打印机、“0”号光电输入机合用一个子通道。假定数据传送期内磁带机每隔 25 微秒发一个字节请求，宽行打印机每隔 150 微秒发一个字节请求，光电输入机每隔 800 微秒发一个字节请求，则这五台设备所需的通道流量为：

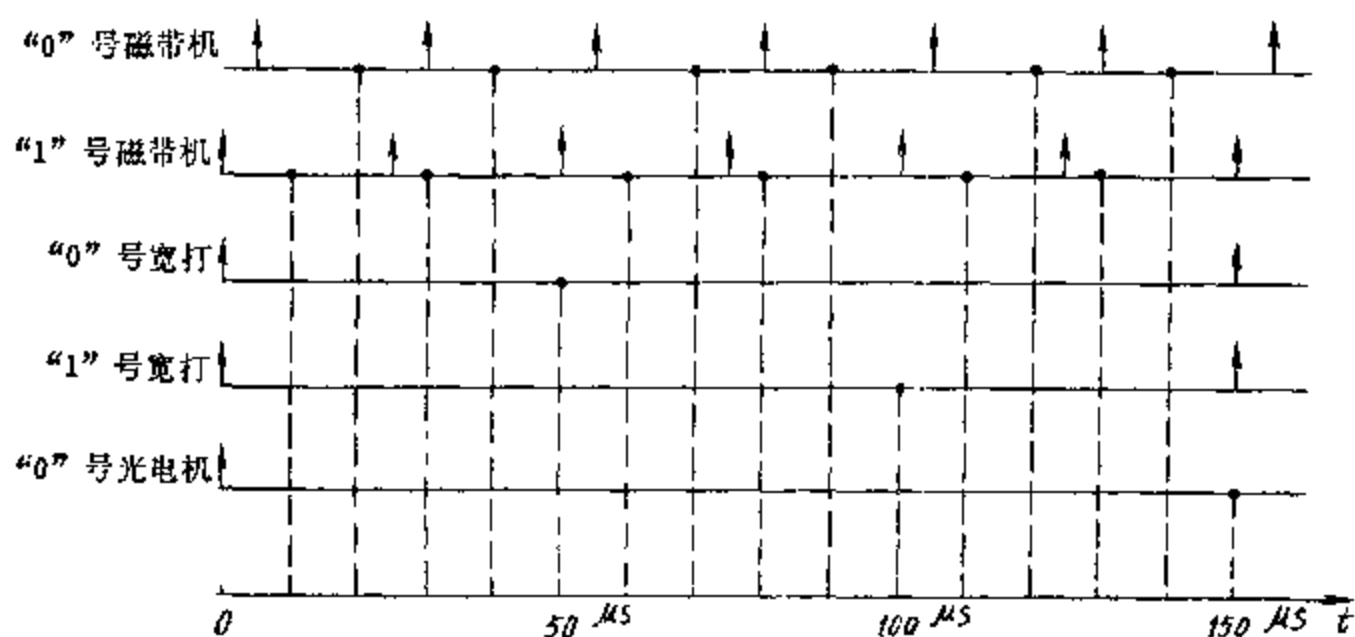
$$f_{\text{byte} \cdot j} = \sum_{i=1}^5 f_{i \cdot j}$$

$$= \frac{1}{25} + \frac{1}{25} + \frac{1}{150} + \frac{1}{150} + \frac{1}{800}$$

$$= 0.095 \text{ 兆赫}$$

就是说，所设计通道的工作周期若为 10 微秒，则可以满足架接设备的流量要求，使各设备请求都能及时得到响应。不会出现丢码现象。

设各台设备的优先权次序为：“0”号磁带机→“1”号磁带机→“0”号宽行打印机→“1”号宽行打印机→“0”号光电输入机。另外，假定“0”号磁带机的请求比“1”号磁带机的请求晚 5 微秒，且“1”号磁带机与其余的三台设备同时发申请。则此通道的工作示意图如图 4.22 所示。看出，果然没有出现丢码。



“↑”表示设备提出申请的时刻
“.”表示通道处理完设备申请的时刻

图 4.22 字节多路通道的工作示意图

4.1-3 通道系统的功能

下面以 IBM370 的通道系统为例，来阐述它的功能。如图 4.23 所示，它是 CPU/主存—通道—设备控制器—外围设备四级结构。

CPU 执行输入输出指令（管态指令）以及处理来自通道的中断，输入输出指令指定通道、子通道及设备。来自通道的中断包括正常执行结束和中途出了故障两种。

通道系统的功能为：

(1) 接受 CPU 发来的输入输出指令，按指令要求与指定的输入输出设备进行联系。

(1) 从主存取出属于该通道程序的通道指令，对命令进行译码，向输入输出设备和设备控制器发送各种命令。

(3) 根据要求,为主存和外设分配和拆卸信息;控制主存与设备之间的数据传送,提供信息传送的通路;指示数据存放的主存地址及传送的字数。

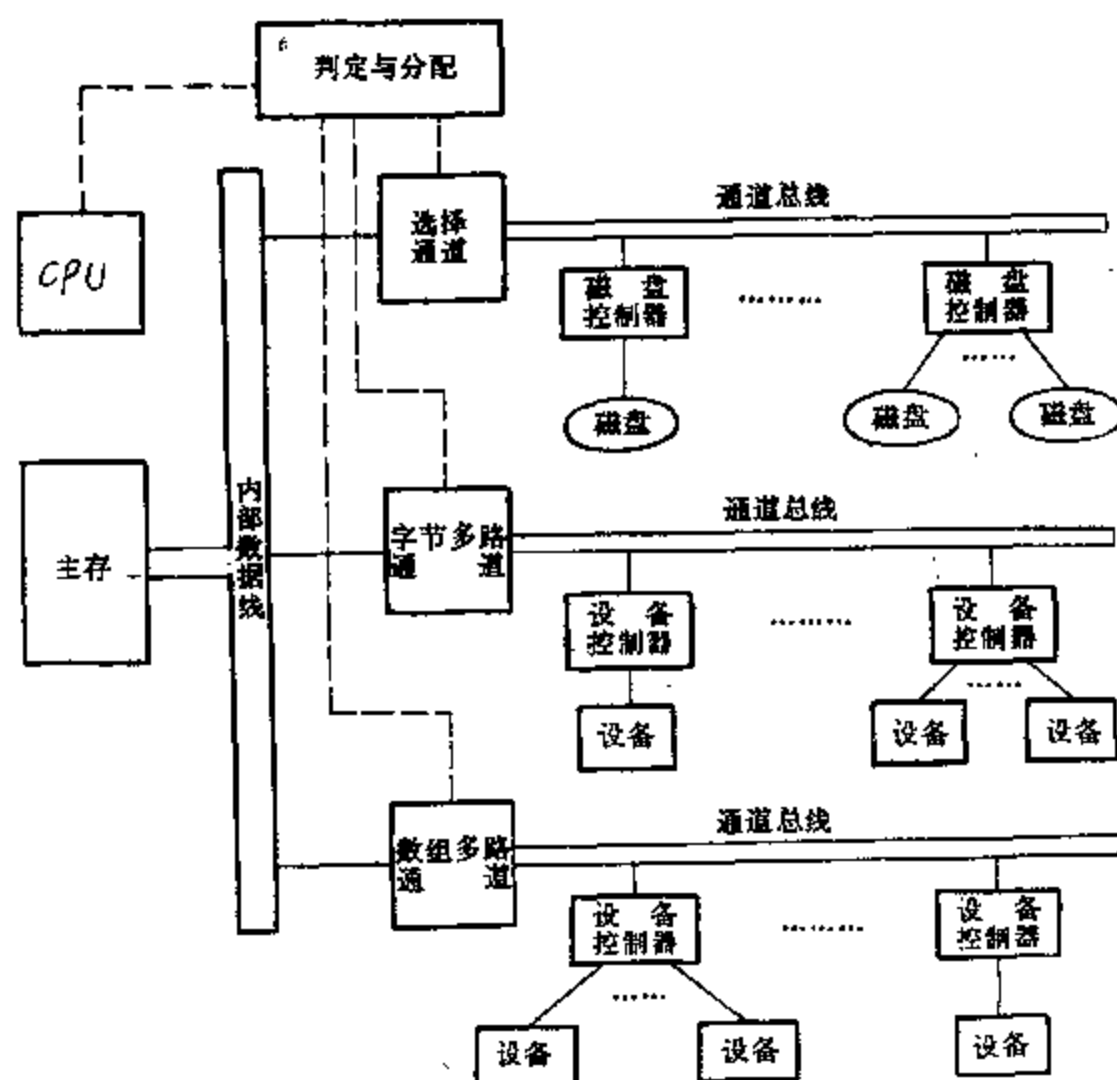


图 4.23 IBM370/165 I/O 结构

(4) 从输入输出设备获得设备状态信息,形成并保存通道状态信息,并可根据需要将这些状态送往主存指定单元。

(5) 将输入输出设备的中断请求及通道本身的中断请求报告给中央处理机。

输入输出控制器实施对设备的控制,设备不同,控制器也不同。但控制器与通道是以标准的接口方式连接,它从通道接受命令,控制设备完成给定的操作,并将各种设备的不同信号转变成符合标准接口且通道能识别的信号,还向通道反映设备的状态。

我国的 DJS-200 系列,也是采用类似的四级连接方案,采用了“字节多路通道、数组多路通道”两种类型。字节多路通道中包含基本多路通道部件和选择子通道部件,均用各自的标准接口与设备控制器相连。数组多路通道除按数组多路通道的基本操作方式外,还可按成组选择方式操作,不过此时由于通道仅为一台设备独占,工作效率要下降。

4.1-4 接口总线

在通道方式的 I/O 系统中,把外部设备控制器与外设之间的接口称为“设备接口”。由于设备的很大差异以及生产厂家的不同,目前这种“设备接口”只能做到在厂家内部或系列内部是标准的。

我们把通道和输入输出控制器之间的接口称为“输入输出接口”。随着操作系统的发展和计算机应用领域的扩大、外部设备的种类和台数的剧增,输入输出接口对计算机的扩展性

和兼容性有着很大的影响。

在输入输出接口上可以采用同步式或异步式总线通讯技术，由于是速度差异很大的各种 I/O 设备共享总线，所以，多采用异步通讯方式。总线的控制技术一般多采用串行链接方式，也有采用独立请求方式的。

总线式标准接口实际上包括接口控制逻辑和接口总线两部分。我们在这里以 IBM360/370 为例进行说明。IBM360/370 通道中的输入输出接口控制逻辑包括若干缓冲寄存器，其基本功能是根据输入输出设备地址进行多路扫描转接。所有连接到设备控制器上的外部设备都可通过控制器对接口总线提出请求。接口控制逻辑按优先次序对各控制器进行扫描响应，由于采用异步互锁的总线通讯方式，可靠性高，并便于多机通讯和远距离终端的非同步通讯。其逻辑功能应是既适用于字节多路通道，也适用于选择通道和数组多路通道。由于采用与通道种类和外部设备物理性能无关的接口技术，因此，只要按接口约定来设计设备控制器，不断出现的新型外部设备都可接到通道上，从而保证了通用性和灵活性。

但是，采用标准接口会使设计复杂。另外，由于采用互锁方式，接口的转接费时，速度慢，特别是在字节交叉方式中，每传送一个字节都要进行三问三答，使接口的流量受到一定限制；对所有连接的设备控制器环行扫描，也影响总线控制的速度。

下面介绍 IBM370 的输入输出接口，它的总线如图 4.24 所示。它大致可分为总线输出线、总线输入线、输出标志线、输入标志线、控制线、扫描线及特殊专用线等七个部分。为了减少线数，采用不同时出现的信息分时使用同一组总线的技术。

总线输出线有九根。其中八根为信息线，一根为奇偶线。信息可能是数据、设备地址或控制命令，这由输出标志线来定义。

输出标志线有三根。由其中的地址输出线和命令输出线分别指明总线输出线上所载的信息是设备地址还是命令字节；还有一根称为服务输出线，对于输入型操作，它的信号升起表明在总线输入线的数据已为通道所接收，这相当于本章 §2 中的“数据接受”线。对于输出型操作，它的信号升起表明外部设备所需的数据已放在总线上了，这相当本章 §2 中的“数据准备”线。这三根标志线是互锁的。

总线输入线有九根。其中八根为信息线，一根为奇偶线。总线上所载的信息可能是数据、设备地址或状态信息，这由输入标志线来定义。

输入标志线有三根。地址输入线和状态输入线的含义与地址输出线和命令输出线相对应。另一根为服务输入线，对于输入型操作来说，它相当于“数据准备”线。对于输出型操作，它相当于“数据请求”线。这三根标志线也是互锁的。

控制线有四根，它们是：

操作输出线，它从通道接到所有设备控制器。它是接口可工作的标志线。当它为低电平

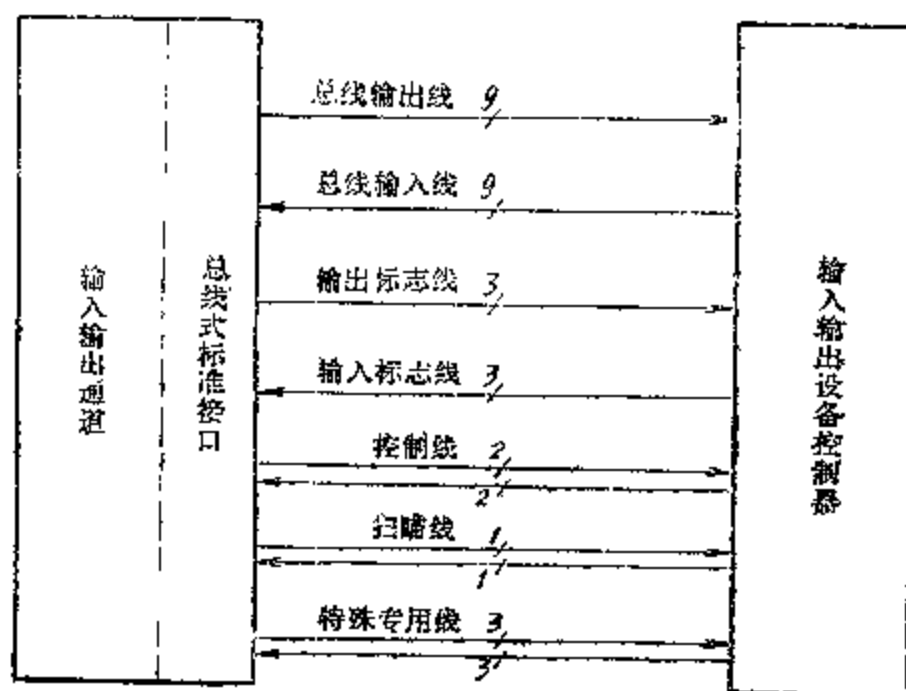


图 4.24 IBM370 的输入输出接口总线

时，接口处于封锁状态；

操作输入线，它从设备控制器接到通道。它为高电平，表示有设备被选中，设备控制器正在与通道通讯；

禁止输出线，它从通道发送到所有设备控制器，主要用来作特殊控制用。譬如说，通道不能接受状态字节、通道来不及处理设备控制器送来的数据、需要进行命令链接等；此时，升高禁止输出线上的电平，禁止信息发到通道；

请求输入线，它是从设备控制器接到通道。它指明设备控制器有状态或数据请求。

扫描线为两根，即扫描输出线和扫描输入线。扫描输出线首先接到优先级最高的设备控制器，再送到次高优先级的，如此一直串行连接到最低优先级的设备控制器，并与扫描输入线连接起来，构成一个“扫描环”，用来选择要求操作的输入输出设备。其功能与前述“总线系统”中所讲的串行链接相同。

特殊专用线有若干根。这些控制线是根据特殊需要而设置的。譬如说，作为时间计量而设置的“时钟输出”线、“计量输出”线、“计量输入线”等；为故障报警而设置的“报警输入”等等。

IBM370 接口总线均采用单向的传输线，因此线数较多。

4.2 通道结构及其操作过程

本小节先介绍 IBM370 的输入输出指令、通道指令和通道程序、输入输出中断和通道状态字；而后，以具体例子介绍通道操作的全过程。

4.2-1 输入输出指令

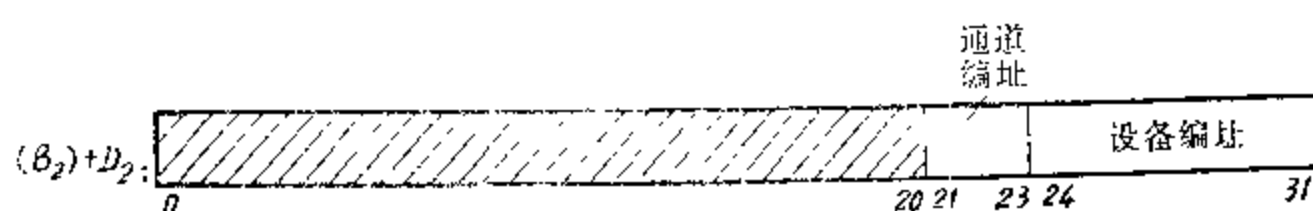
输入输出指令是中央处理机用来控制输入输出操作的指令。指令的格式和类型可根据整机系统的要求而定。IBM370 和我国的 DJS-200 系列都采用五条输入输出指令，这些指令均定为管态指令，目态下禁止使用这些指令。

输入输出指令之所以定为管态指令主要有两个原因。其一是若输入输出指令是目态指令，则用户就可以用这些指令来“读取”存储在 I/O 系统中但规定不允许他读出的那些信息；另外，用户也可以通过“写入”破坏其它用户所存的内容。其二是用户也不愿意用目态的输入输出指令来自己编写输入输出程序。例如，在 PDP-11 中，它的输入输出指令虽然比较简单，但程序员想要用它来编制磁盘的输入输出程序，那还是相当麻烦的。因为，一条磁盘命令执行的结果，可能会出现各种各样的错误。例如，读定时错、在前一命令完成之前启动磁盘输入输出、在写操作后的写校验错、不存在的磁盘编址等等。为处理每一种情况，程序员都必须编写处理例程序，在编写时还必须记住有关的状态信息。所以，目前大部分机器的输入输出指令不属目态指令。

由于输入输出系统是由几级组成，系统中的每一级都可能处于空闲、忙、断开等状态，在执行输入输出指令期间，通道就要根据通道、设备控制器、设备等各级的状态形成条件码反映给程序，作为本次执行输入输出指令的结果。管理程序根据其送来的条件码，确定程序的去向，并以此释放 CPU。

IBM370 的输入输出指令采用 $(B_2) + D_2$ 所形式的地址作为通道及设备编址，若其最大

通道数为8，最大设备编址为256。其格式如下表示：



后来 IBM3030 的通道编址字段扩展到 8 位。

IBM370 的输入输出指令原来有五条，即启动 I/O 指令 (SIO)、查询通道指令 (TCH)、查询 I/O 指令 (TIO)、清除 I/O 指令 (CLRIO) 和停止 I/O 指令 (HIO)。后来，随着系统性能的提高，IBM3030 系列，使用了九条输入输出指令，即除了上述五条外，增加了启动 I/O 快释放指令 (SIOFR)、停止设备指令 (HDV)、存贮通道记录区数据指令 (STIDC) 和清除通道指令 (CLRCH)。

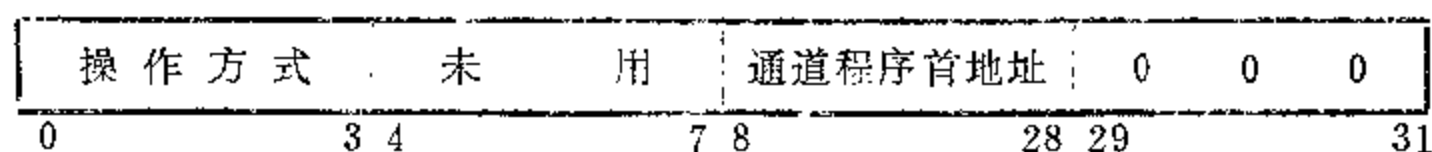
一、“启动 I/O”指令和“启动 I/O 快释放”指令

这是主要的中央 I/O 指令，它是用来启动指定的通道、子通道及外设以执行管理程序所要求的输入输出操作，执行时要检查指定通道、子通道及设备状态，据此形成条件码，用它把“启动 I/O”指令执行的情况通知管理程序。

有了“启动 I/O”指令，再加上中断处理，就可以控制对所有连接在通道上的设备的输入输出操作。其它的输入输出指令都是为执行辅助操作而设计的，以便能更好地发挥通道及输入输出设备的作用，以及使操作系统能更好、更灵活地管理输入输出设备。

在启动输入输出后，若启动成功，中央处理机与通道可以“并行”操作，通道按操作系统编制的通道程序执行各种操作。

在主存固定单元中设有“通道地址字” (CAW)，其格式如下：



在“启动 I/O”指令之前，必须在 CAW 中置入通道程序首条指令的地址，同时还要指明操作方式，这些都由操作系统置入。“操作方式”仅对数组多路通道有意义，它有数组多路方式和数组选择方式之分。

当执行“启动 I/O”指令时，要从主存中取出 CAW，而后再按 CAW 给出的通道程序首地址从主存中取出第一条通道指令。

“启动 I/O”指令的流程图如图 4.25 所示。

在发出“启动 I/O”指令后，如果通道、子通道是空闲的，且设备以全“0”字节回答通道，表示设备接受并执行这个命令，则启动成功。此后，设备在通道的控制下，完成所要求的动作。

IBM3030 系列增设“启动 I/O 快释放”指令，是因为原来 370 的“启动 I/O”指令，在取第一条通道指令及启动外部设备的过程中，CPU 是踏步等待的，而对这一条新指令，在向主存固定单元取出通道地址字，判完通道地址字的格式无误后，即可用条件码回答，释放 CPU。以后的通道操作可和 CPU 并行进行，以减少 CPU 的等待时间。

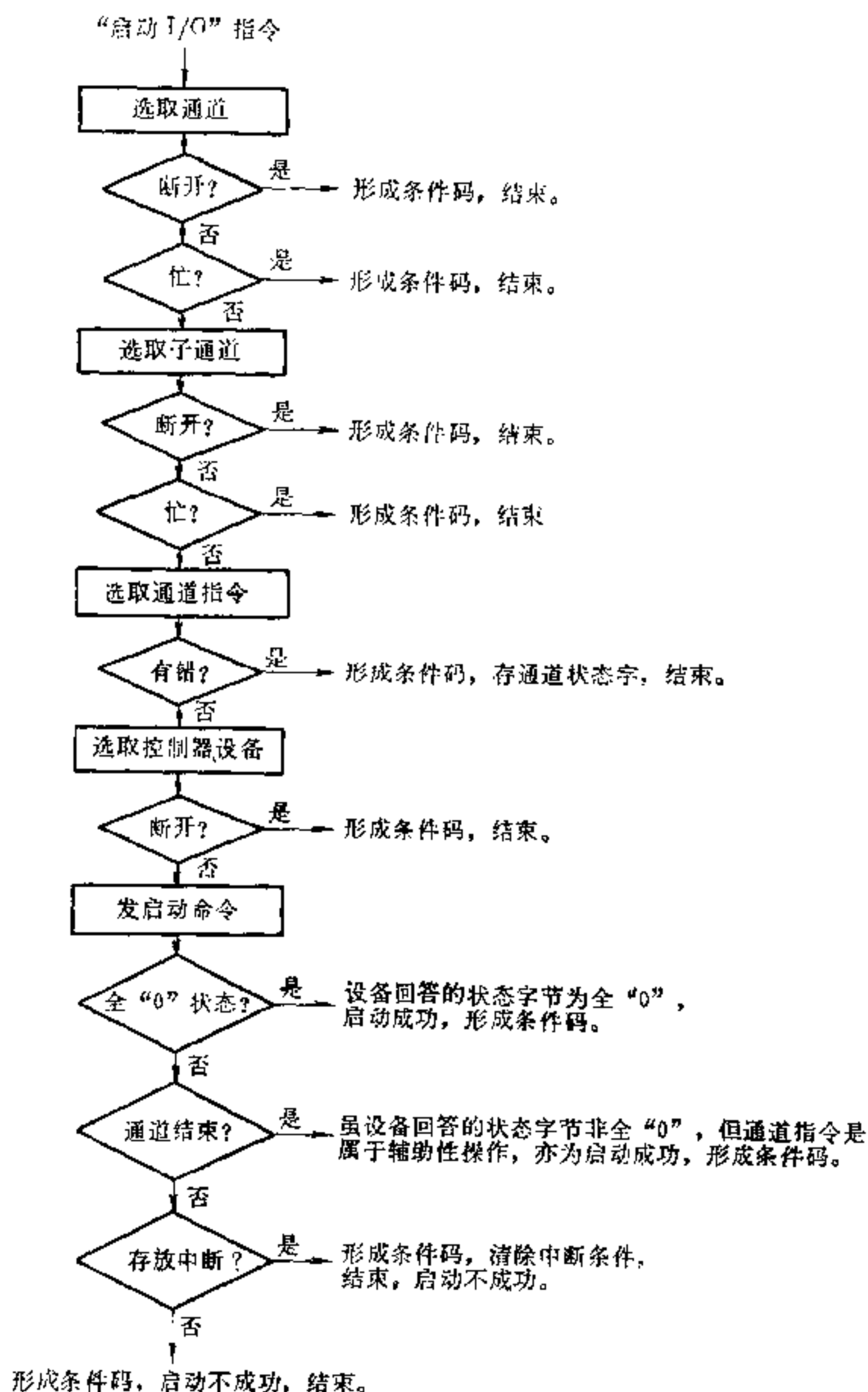


图 4.25 “启动 I/O” 指令流程图

(注: 这里的结束, 表示释放通道)

二、“查询通道”指令

它用于快速查询通道的现行状态。它只查询指定通道的忙/闲、是否存中断或断开的情况, 不访问子通道和设备。查询结果以条件码回答 CPU。其指令流程图如图 4.26 所示。

然而, 由于一个字节通道通常含有多个选择子通道, 每个选择子通道又都有其标准接口, 若只查询通道的忙/闲, 并不能真正了解通道中哪个接口处于忙, 哪个接口正空闲着。显然, 若寻址到子通道, 则能准确地知道指定的子通道或指定的输入输出接口是否处于忙。我国的 DJS-200 系列机就是如此安排。

三、“查询 I/O” 指令

它的功能主要用来查询指定通道、子通道和输入输出设备的状态，形成条件码，为程序提供比较详细的输入输出操作执行情况。当指定子通道或设备上有中断请求时，则取回中断请求；利用这一点，通过在程序内的不同点设置查询 I/O 指令，就相当于可改变各个输入输出设备的中断优先权。

四、“停止 I/O” 指令、“停止设备” 指令、“清除 I/O” 指令

这三条指令都是用于停止输入输出操作，设置后两条指令是因为靠“停止 I/O” 指令要停止在数组多路通道中的成组操作很麻烦，用这二条指令就可以快速地断开通道与输入输出控制器通路，让位于更高优先权的输入输出操作，同时也便于由 CPU 程序停止指定设备的现行操作。同样，这些指令根据通道、子通道和设备的现行状态，形成条件码。

五、“存贮通道记录区数据” 指令和“清除通道” 指令

这二条指令都是 IBM3030 系列新加的。前者主要用于在通道发现硬件故障，例如发现通道控制错、接口控制错或电源故障等，可用它将错误的大致位置存在输入输出通讯区中，供程序分析使用。后者用于清除通道状态，例如清除接口控制错。以免由于接口控制错而造成子通道错误蔓延。

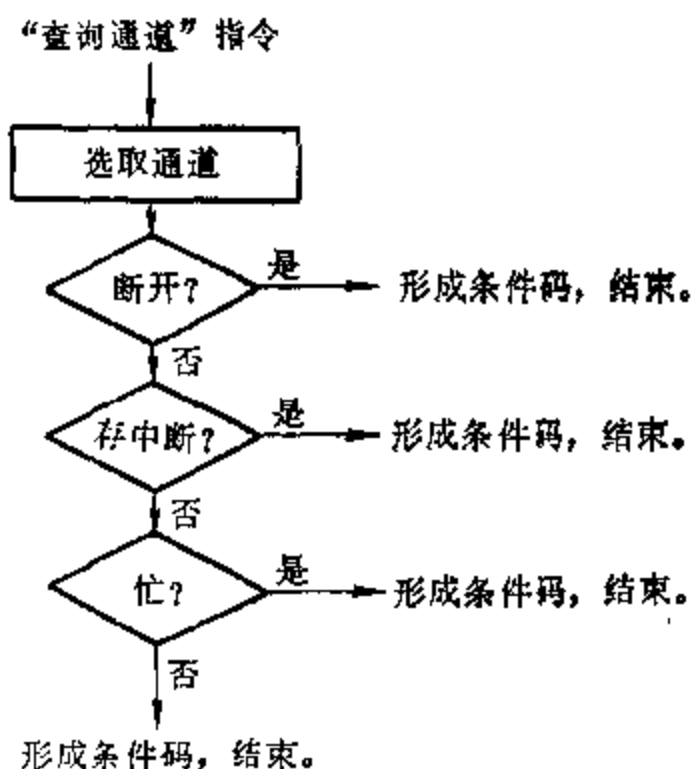


图 4.26 “查询、通道” 指令流程图

4.2-2 通道指令和通道程序

通道指令亦称通道控制字 (CCW)，它也存放在主存中，它是由通道取出并由通道执行的控制字。每条通道指令为输入输出设备规定一定的动作。由一条或几条通道指令链接组成通道程序，完成 CPU 的一条输入输出指令所要求执行的操作。

一、通道指令

通道指令至少包括四个字段：命令、数据地址、控制特征和计数值。不同的通道，通道指令的格式不完全相同。下面是可能的一种形式。

命 令	数据地址	特 征	0 0 0	未 用	计数值
0	7 8	31 32	36	40 47 48	63

命令字段给出通道执行的输入输出操作。命令码可分成补充位和基本操作位两部分。基本操作位决定了通道及设备操作的基本类型；补充位进一步定义具体的设备所需的不同动作，它因设备而异，通道不加识别，通道只识别基本操作位。下面主要介绍基本操作。IBM 370 采用七种基本操作命令：读、反读、写、控制、断定、通道转移及不操作。

读命令，也称输入命令。它命令指定的输入输出设备向数据地址字段所指明的主存数据（缓冲）区传送数据，传送的字节数由计数值字段指明。

反读命令，使某些输入输出设备，如顺序存取的磁带在反转时进行读取，由设备向主存

按地址递减方向输入数据。除地址递减外，其它与读命令相同。

写命令，也称输出命令。与读命令类似，只是数据传送方向相反，是从主存送往输入输出设备。

控制命令，不是用于数据传送，而是控制设备进行非数据传送型操作，为读、写命令的执行作准备。

断定命令，其基本功能是将设备的状态信息传送到主存指定单元，以判定设备出错的原因和部位。例如在设备出错时，设备状态字节中的“设备出错”位变成“1”，从而形成中断。管理程序为进一步了解设备出错的情况，就发断定命令，把反映设备出错情况的断定字节送往数据地址字段指定的主存地址。

通道转移命令，对于一道通道程序中的各条通道指令，如果在主存中不是按地址顺序存放，则地址不相邻的两条通道指令可用通道转移命令加以链接。这时，数据地址字段为下一条通道指令的主存地址。当通道转移命令与设备状态字节中的转移标志位连用时，可以控制通道程序的条件转移，数据地址字段作为转向去址。通道转移命令仅在通道中执行，与设备无关。

数据地址字段，一般采用直接地址，但 IBM3030 是采用间接地址。

控制特征字段又称为标志码字段，主要用来定义通道程序的链接方式或定义该通道指令的操作特征。IBM中有数据链（CD）、命令链（CC）、封锁错误长度（SLI）、封锁写入主存（SKIP）以及程序控制中断（PCI）等五个特征。

综上所述，通道中可以执行读、写、控制、反读、断定状态和通道转移等六种输入输出操作。执行断定状态操作时设备送出其当前的状态信息和接口中的异常。执行通道转移操作时，下条通道指令取自现行通道指令给出的主存地址，而不是按地址顺序取。看出，通道指令和 CPU 指令在原理和概念上是一样的，只是通道指令的结构格式和功能专用些而已。

二、通道程序

通道程序由一条或几条通道指令组成。如果由几条通道指令组成，就称通道指令链。前已说过有命令链和数据链二种。命令链由几条命令码不同的通道指令构成，这就能使 CPU 只用一条输入输出指令就能控制输入输出设备执行需几个命令才能完成的操作。数据链是由几条具有同样命令码，但对应不同主存数据区的通道指令构成，这用于使数据从主存送往设备或从设备送往主存的过程中，控制数据的更合理分布。

下面以二道不同的通道程序为例来说明。

例 1 通道程序只由一条通道指令组成：

命 令	数 据 地 址 (十六进制表示)	控 制 特 征 (二 进 制 表 示)					计 数 值 (十进制表示)
		CD	CC	SLI	SKIP	PCI	
读	30A0	0	0	0	0	0	500

表示从磁带存储器（由命令码中的补充位给定）读出长度为 500 个字节的完整数据区，送往主存（30A0）₁₆ 起的 500 个单元中。

例2 通道程序由三条通道指令组成：

命令	数据地址 (十六进制表示)	控制特征 (二进制制表表示)					计数值 (十进制表示)
		CD	CC	SLI	SKIP	PCI	
1 读	30A0	1	0	0	0	0	256
2 未用	未用	1	0	0	1	0	32
3 未用	6000	0	0	0	0	0	4096

表示从磁带存储器（由命令码中的补充位给定）读出 4384 字节的数据区，其中 1—256 字节读到主存从 $(30A0)_{16}$ 地址开始的区域，257—288 字节由“SKIP”为“1”控制不存入主存，而 289—4384 字节读到主存从 $(6000)_{16}$ 地址开始的区域。这三条通道指令用数据链特征 CD 链接成通道程序。

使用相应的通道程序可以执行各种类型的操作和数据传送，如数据重新分配读操作（把一个固定的数据区读到主存几个互不相连的区域）；有选择的读操作（把数据区各个被选的部分读到主存）；无数据传送的输入输出操作（如反向转动或磁带反绕）等等。

通道程序在主存中必须保留到执行完毕。通道每次只从主存中取一条通道指令，并将其保存在子通道中，用以控制设备的现行操作，直到本条通道指令全部功能执行完毕，才去主存取下条通道指令。

4.2-3 输入输出中断

在通道程序执行过程中，如果遇到某通道指令的数据链和命令链特征位均为零，则表示本条通道指令是通道程序中的最后一条了。当该通道指令动作完成时就要发 I/O 中断通知 CPU 程序，告诉它本通道程序已顺利结束。另外，如果在命令链链接期间，通道识别了任何一种错误，或设备和设备控制器发现了某种错误，以及当出现链中止等情况时均要发中断通知 CPU。

输入输出中断是输入输出系统与 CPU 联络的基本手段。不论是 I/O 系统已正确无误地完成操作，还是因为设备、控制器或通道出现故障（损坏或掉电）、错误（数据传送错、控制错、链接错、执行错误的 I/O 指令）等等，都要求 CPU 进行处理。因此，通过 I/O 中断应能告知 CPU：输入输出操作正常结束；操作提前中止的原因；通道程序沿链推进的情况；某台设备有外界请求；I/O 系统出现其它异常等等。

CPU 响应 I/O 中断后，首先将发中断的通道和设备的地址，作为中断码写入旧 PSW 中保存起来，再将指明这些状态信息的通道状态字送入主存固定单元，供中断处理用，然后取出新 PSW，执行中断处理。

各通道的中断优先次序可以根据通道架接的位置或通道的序号来决定，而同一通道中各中断源的排队，可采用“串行链”方式，即按形成中断条件的输入输出设备在输入输出接口上架接的位置远近来决定。当然也可使用启动 I/O 指令或查询 I/O 指令来改变中断优先权。

4.2-4 通道状态字

通道状态字 CSW由通道、设备控制器和设备的状态，以及下一条通道指令所在主存单元的地址组成，存贮在主存的固定单元。它可经 I/O 中断写入主存，也可以在 I/O 指令执行期间，直接存入主存。可以把它看成是输入输出中断码的补充。

通道状态字随不同的约定而异，但一般有四个字段组成，即保护键、通道指令地址、状态、计数值。下面是其中的一例：

保 护 键	通道指令地址	通道与设备状态	计 数 值
0	7 8	31 32	47 48 63

保护键字段包括子通道的四位访存访问键(参看下一章 § 7)。通道指令地址字段包含下条通道指令的地址。状态字段包括设备状态和通道状态二部分。计数值字段，指出上一条 CCW 的操作结束后剩余的数据字节个数。

设备状态字节是由设备控制器和设备产生，并通过标准接口送来，它对控制 I/O 的操作有重要作用，因为通道的操作在很大程度上受设备状态控制。一般它应包括有：注意、忙碌、通道结束、设备结束、设备出错、转移等。“注意”是由 I/O 设备识别的，对程序有意义的异步信号，它由程序进行解释，一般多用于人工请求输入，用于人——机联系或其它外围呼叫的场合。例如，控制设备开始执行手动操作时，就要发“注意”。“忙碌”表示设备或控制器正忙，不能执行命令。“通道结束”表示设备已不再占用子通道，设备与通道间信息传送已经结束。“设备出错”表示设备或控制器发现程序出错或设备故障，但具体错误情况需借助断定字节指明。“转移”用于反映通道程序的跳跃或其它转移。

通道状态字节包括诸如通道数据错、通道控制错、接口操作错、传送长度错、保护出错、链出错等等。

中断处理程序在分析判断时，还可利用断定命令到设备控制器取断定字节，它存有更详细的设备、控制器状态信息。

4.2-5 通道的工作过程举例

本小节以控制光电输入机的输入过程为例，阐述通道的工作过程。

一、从广义指令到“启动 I/O”指令

我们说过，“启动 I/O”指令应属管态指令，那用户又怎么能调用外部设备呢？他是通过使用广义指令来使用外部设备的。广义指令的一个用途是实现目标程序和管理程序之间的联系，目标程序对外部设备的使用要求，通过广义指令由管理程序解释执行。

广义指令是由访管指令和若干参数组成的。访管指令为目态指令，广义指令操作码实质上即为对应此广义指令的管理程序入口。管理程序根据广义指令中所给的参数，编制好通道程序，并存放在内存通道程序缓冲区。经各种有效性检查后，再将通道程序的入口置入主存“通道地址字”单元。之后，管理程序就执行“启动 I/O”指令，在取出第一条通道指令后，通道按此指令选择并启动指定设备，这个时期称为通道开始选择设备期。

二、通道开始选择设备期

这期间的工作过程如下：

(1) 通道将“启动 I/O 指令”指明的设备地址（本例为某光电输入机号），经标准接口放到“总线输出线”上④，升起地址输出标志线，表示总线输出线上所载的信息是设备地址。架接在接口总线上的各设备控制器把自己的地址与总线送来的指定设备地址进行符合比较；

(2) 通道用“扫描输出”线进行扫描。当扫到地址符合的设备（光电机）时，其控制器经“操作输入”线回答，告诉通道已找到所要的光电机；

(3) 设备控制器将其地址经“总线输入”线送回通道⑤；

(4) 通道对设备送回的地址进行奇偶检查，并将其与发送的地址进行比较⑥，如相符，则通道把通道指令的“命令码”经总线输出线送往该设备⑦，同时升起命令输出标志线，表示总线输出线上给出的是命令字节；

(5) 设备控制器收到命令并判断正确后，把“设备状态字节”经总线输入线送回通道⑧，如果命令可被接受，则用全“0”状态字节来回答；

(6) 通道收到设备状态字节后，用“服务输出”回答。另外，通道根据送来的全“0”状态字节形成条件码，告诉中央处理机⑨，此次启动成功。

至此，通道开始选择设备期结束，进入数据传输期。从这时起，通道与中央处理机开始并行操作。

看出，对 IBM370 的总线系统，启动一台外部设备，需很多次进出接口总线。显然，

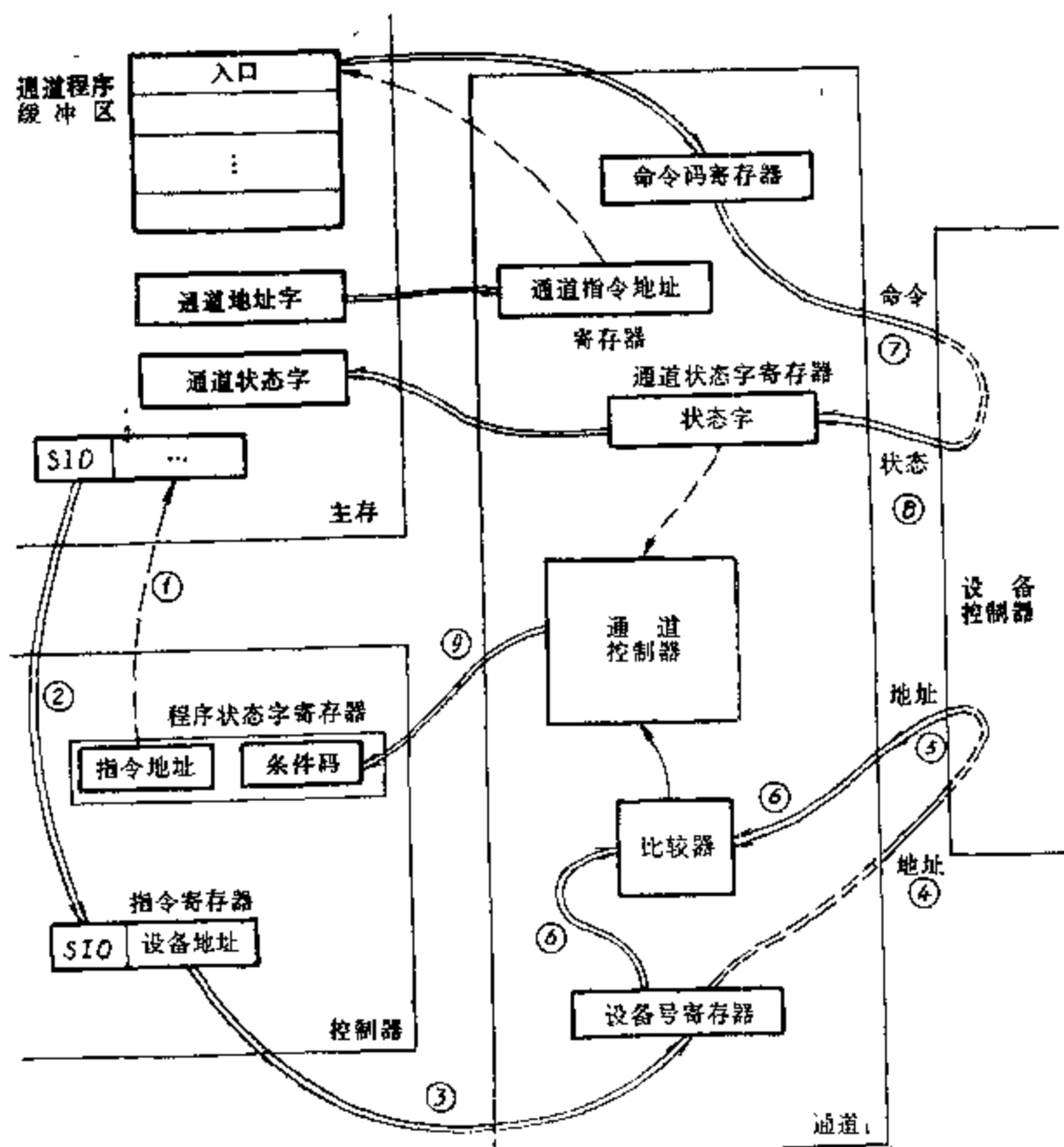


图 4.27 通道开始选择设备期信息流示意图

采用异步双向互锁控制方式所带来的可靠性提高是以时间损失为代价的。而在此期间，CPU 只能一直处于白白等待状态。这种状况，对数组多路通道更为严重，这就是为什么 IBM3030 系列增设“启动 I/O 快释放”指令的原因。

上述通讯过程的信息流示意图和接口时序图表示在图 4.27 和图 4.28 上。

三、通道数据传送期

数据传送期的任务是在指定的输入输出设备被启动后，按设备的要求，实现设备与通道，通道与主存之间的数据交换。由于本例是光电输入机，所以是以字节交叉的方式进行传输，在交换一个字节后，即释放通道、断开接口，供别的设备使用。

现在，是通道等待传送数据的到来，即由设备控制器发出使用通道的申请，所以是从设

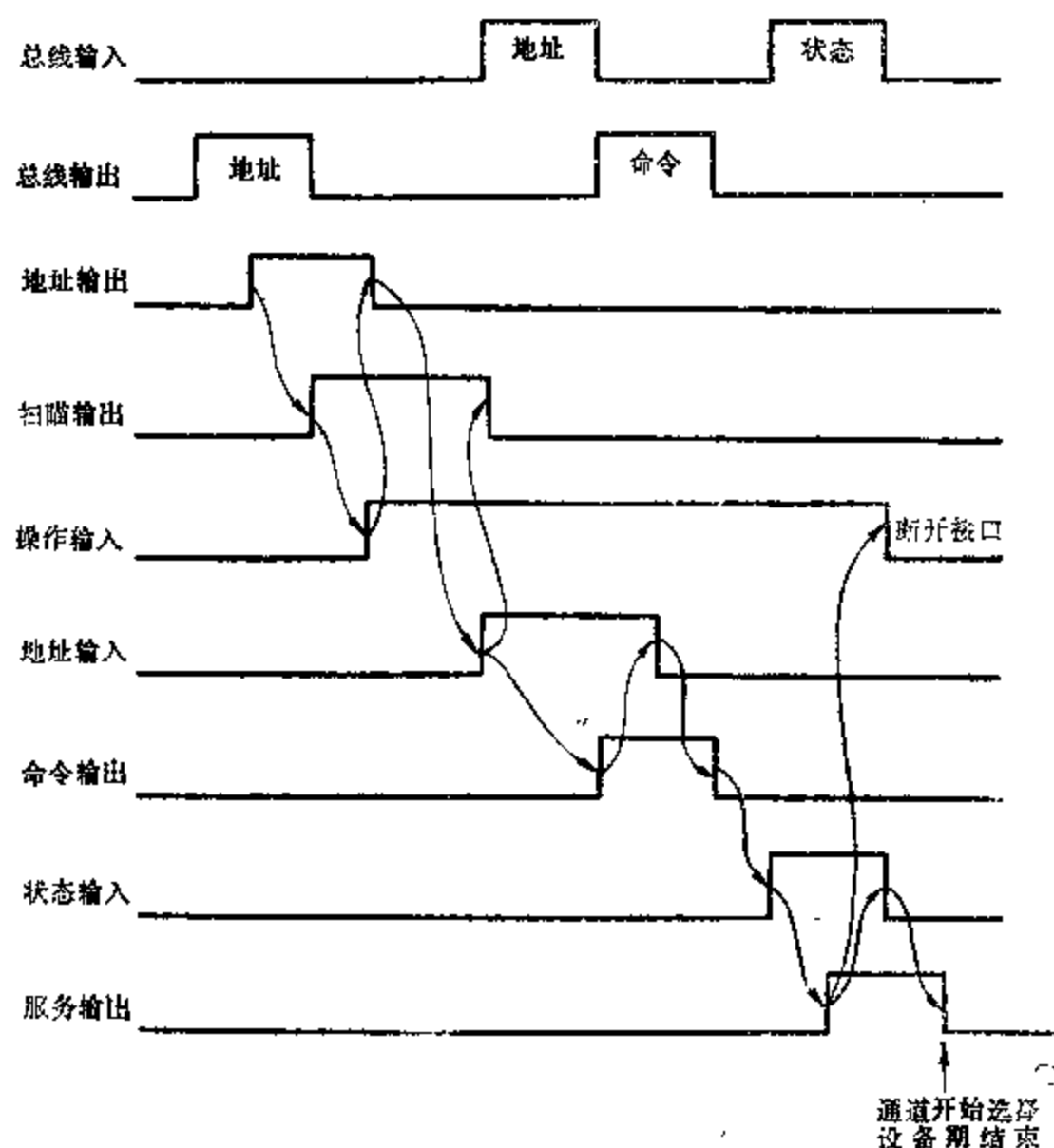


图 4.28 通道开始选择设备期的接口时序

备控制器发请求输入信号开始。其操作顺序如下：

(1) 设备启动后，在第一排数据孔信息经光电转换送至设备控制器数据缓冲器后①，设备控制器即发“请求输入”信号。通道检测到该信号，就发出“扫描输出”信号，按扫描环优先次序选择，在选上光电机时，它的设备控制器发“操作输入”信号，并将其“设备地址”经“总线输入”送回通道；

(2) 通道检查送来的地址，确系指定设备，则经“命令输出”线回答；

(3) 设备控制器将缓冲器上的字节信息放到“总线输入”线上，并发出“服务输入”，表示数据已放在总线上③；

(4) 通道接收输入的字节数据，把它放入字节装配寄存器，并用“服务输出”回答“服务输入”；

(5) 设备控制器收到“服务输出”回答信号后，知道数据已被接收，就降下“服务输入”，表示一个字节已传送完，断开接口，让总线为别的设备数据传送用。

当该设备的第二排带孔经光电转换送入设备控制器时，设备控制器重复上述申请、传送过程。

对于通道，在装配满一个整字时，就传送到缓冲寄存器④，接着就需要申请一个“主存周期”，把数据存入主存⑤⑥⑦。

数据传送期的数据流通路控制示意图及接口时序图示于图 4.29 和图 4.30。

每传送一个字节，“字节计数值”减“1”，当减至“0”时，表示本次使用该光电机结束。然而，过程的结束可能有三种情况：通道先于设备结束，即当通道使计数值已减至“0”，但设备未发现结束，仍请求交换数据；设备根据其物理记录区的边界及其它标记表示传送结束；通道和设备同时发现结束。这样，对于前两种情况，可判定发生了交换字节长度错误（这也可能是由于主存规定的记录缓冲区过小或过大）。此时设备应被强制结束，通过存“通道状态字”再去分析。

在数据传送期之后为数据传送结束期。

四、数据传送结束期

它的操作过程如下：

(1) 当光电机最后一排结束孔标志送至设备控制器时，它发“请求输入”。通道继而发出“扫描输出”，选到此光电机后，设备控制器发“操作输入”信号。同时，将设备地址经“总线输入”线发往通道；

(2) 通道识别地址无误后，发“命令输出”信号来回答“地址输入”；

(3) 设备控制器将状态字节放在总线输入线上，并上升“状态输入”线电平，指明总线上放的是状态字节，而不是数据；

(4) 通道接收状态后，用“服务输出”回答“状态输入”；

(5) 设备控制器收到“服务输出”信号后，断开接口。

结束期的时序如图 4.31 所示。

通道根据送来的状态信息确知输入已经结束，就发 I/O 中断告知 CPU，使 CPU 第二次进入管理程序，进行输入输出中断处理。当管理程序根据通道状态字判明确是正常交换结

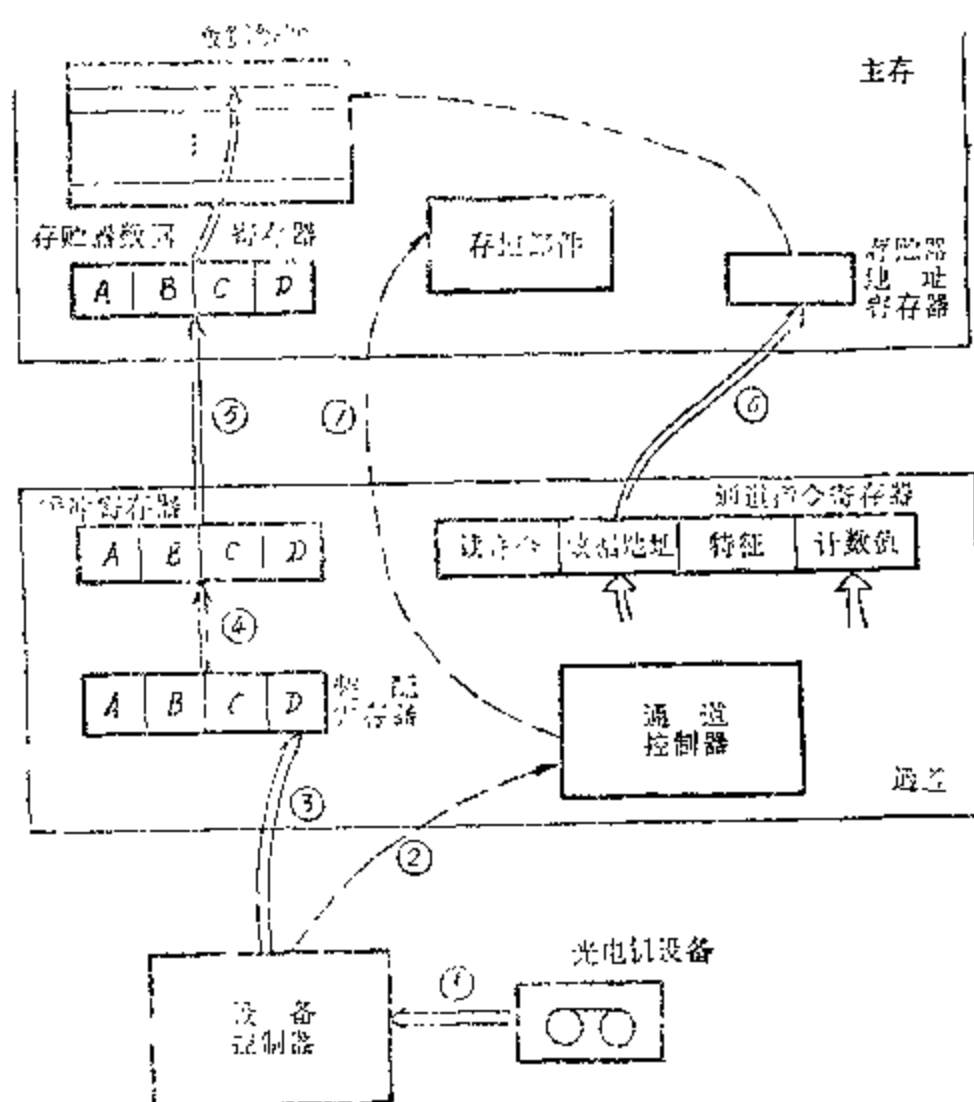


图 4.29 数据从设备到存贮器的传送

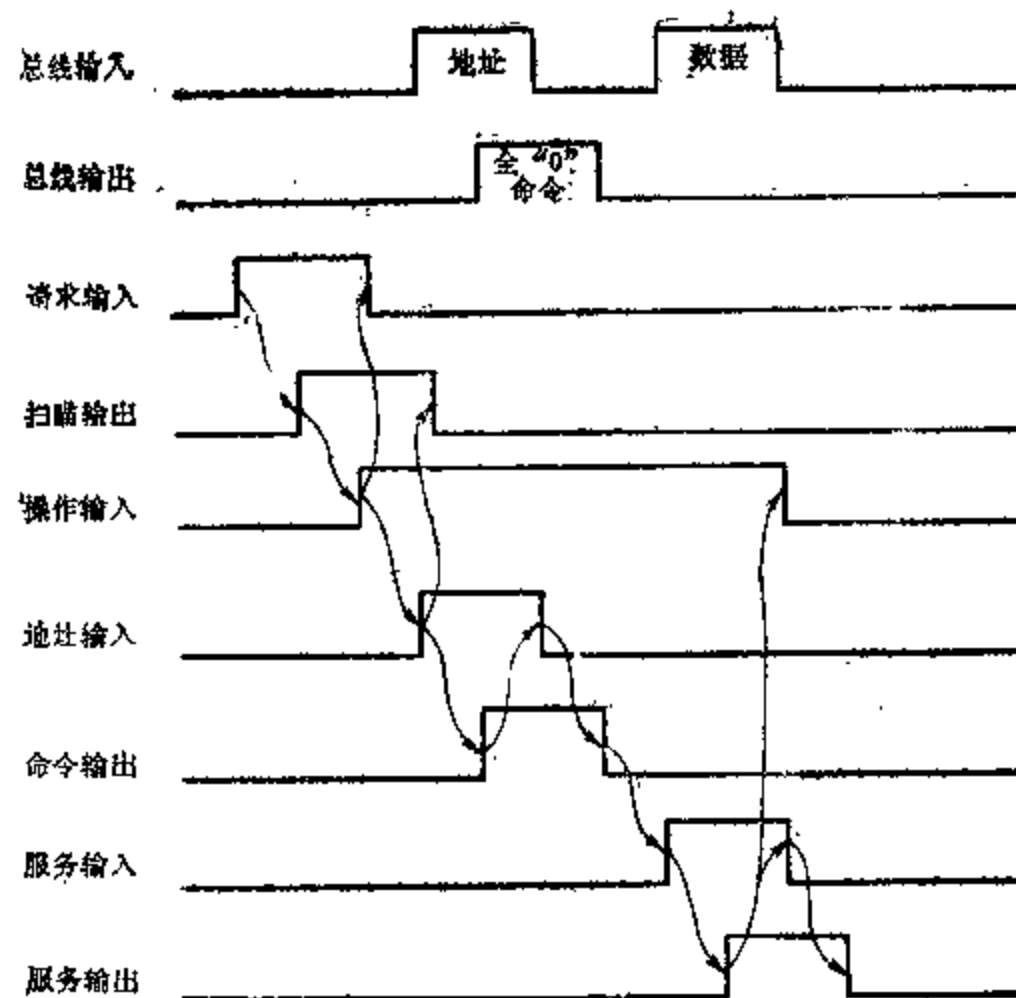


图 4.30 数据传送期接口时序图

束时，它进行登记等善后处理后又转回目标程序继续进行。至此，此次组织的光电输入才全部结束。

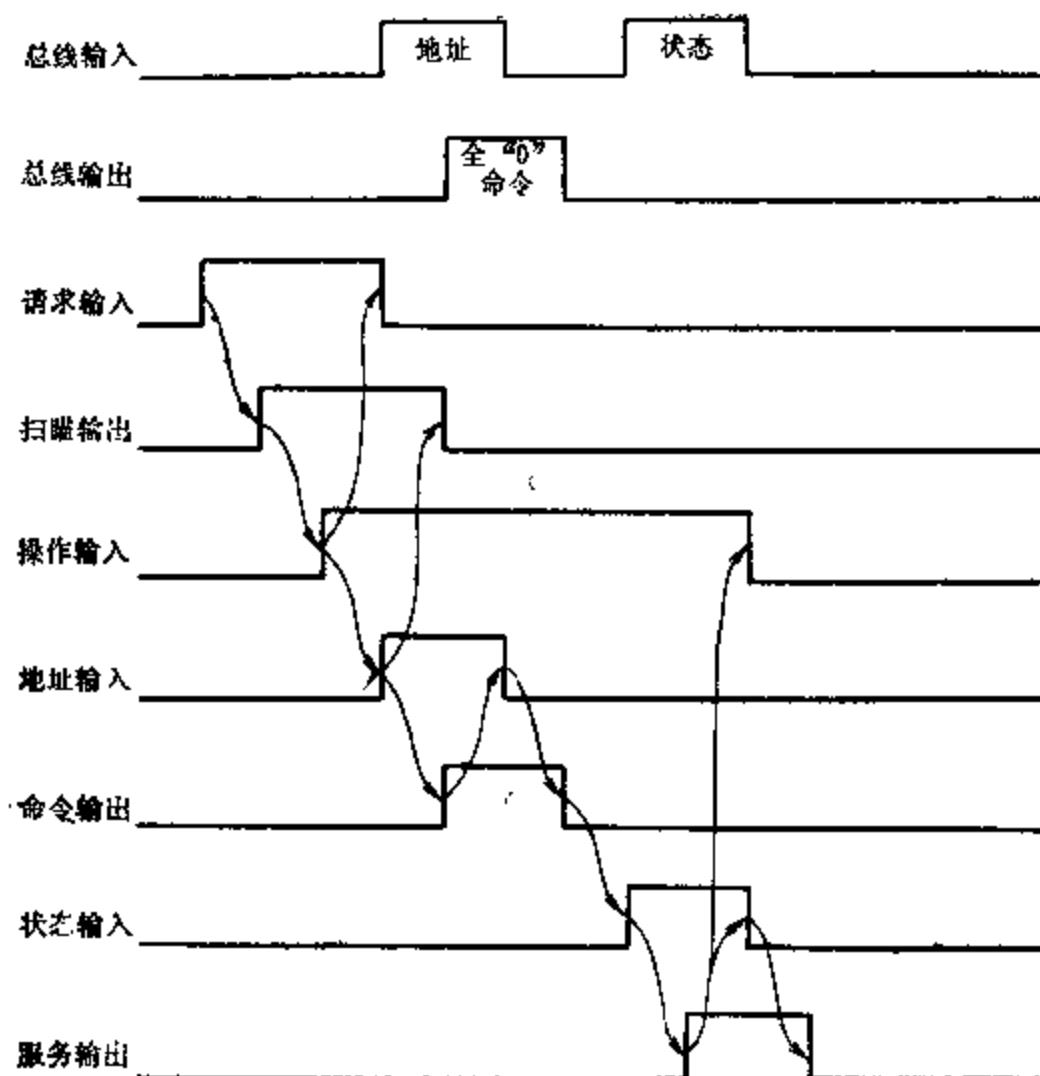


图 4.31 结束期时序

现在，我们再把执行一条光电输入广义指令的主要过程概括地用图 4.32 来回顾。

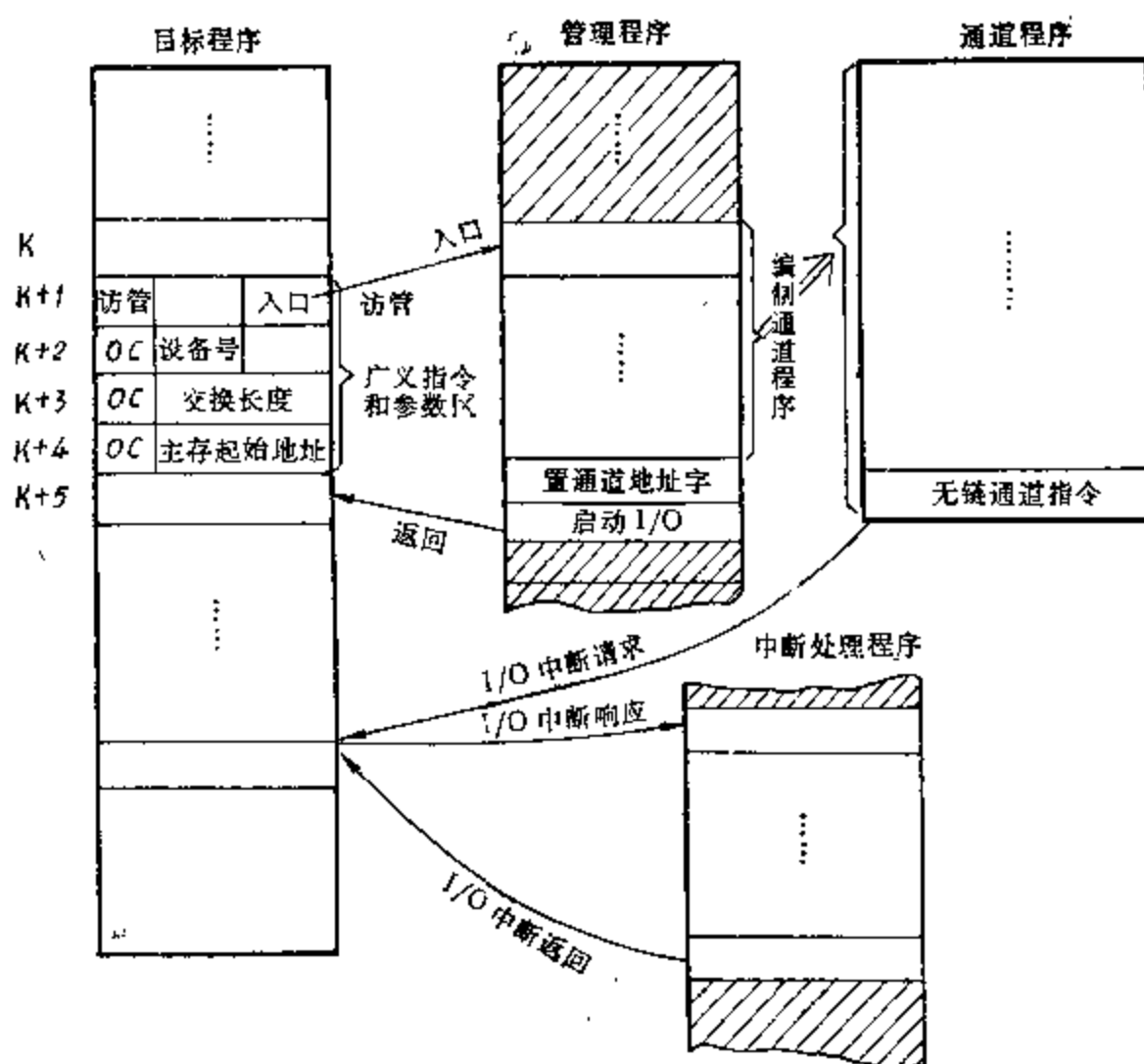


图 4.32 执行光电输入广义指令的主要过程

这个例子，我们虽介绍的是字节交叉方式的输入输出过程，对于选择方式和数组多路方式的信息交换也基本类似。主要不同点是：

- (1) 在通道开始选择设备期，如果传送的通道指令的命令码是数据传送型，为减少接口在工作过程中的转接时间，在选择期结束后不断开接口；
- (2) 在数据传送期，不需要象字节交叉方式那样进行接口选择和断开，以提高接口的传送速率。

前面说过，IBM370 在通道开始选择设备期结束后，通道与 CPU 就可并行工作。此时 CPU 在执行目标程序中可能又会遇到另一条启动外部设备的广义指令，则 CPU 又将执行“启动 I/O”指令，从而实现多道通道程序的并行执行。

§ 5 外围处理机

5.1 基本概念

综上所述，采用通道结构的输入输出系统大大简化了 CPU 对 I/O 操作的控制，基本上实现了 CPU 程序与设备的操作控制之间的并行执行，减少了设备向 CPU 请求中断的次数，从而提高了 CPU 的运行效率。然而，还存在如下问题：

- (1) I/O 传送的前处理和后处理中还得占用 CPU 的时间。
- (2) 通道本身不能解决设备或通道出错以及异常情况的处理，当发生这些情况时，需通过输入输出中断由 CPU 来处理。

(3) 通道结构在传送数据时只能对传送的数据进行装配和拆卸处理,而码制变换、格式处理、整个数据块的校验等还需由 CPU 完成。

(4) 文件管理、设备管理等操作系统的工作,通道结构更是无能为力。而这些输入输出管理工作随着多道程序同时运行的作业数的增加以及系统软件本身的复杂化愈趋繁重。

具有外围处理机的系统,则可以较好地解决上述问题。

在 IBM 发展通道结构 I/O 系统的同时, CDC 和 Burroughs 等公司则是发展外围处理机结构的 I/O 系统。这种外围处理机的结构更接近于一般处理机,或者就是选用已有的通用机。采用外围处理机结构除了使 CPU 能摆脱对输入输出操作的控制以外,还因外围处理机功能的加强能简化设备控制器。有的系统还利用外围处理机作为诊断、维修、系统工作情况显示和人—机联系之用。外围机基本上是独立于主处理机工作的。

就现有的外围机 I/O 系统来看,按外围机与中央处理机共享主存与否,可以有两种连接方式。

一、共享主存的连接方式

中央处理机和外围机都能对主存进行直接的读、写访问,如 CDC-CYBER 系统、ASC 系统和 Burroughs B-6700 等都属于这种结构。在这种连接方式中,外围机内的存贮器通常不大,外围机执行的例行程序平时存放在主存内,需用时通过加载或更换复盖等形式把它调入外围机存贮器中,实现了各外围机对例行程序的分享。例如 CDC-CYBER 系统的外围机存贮器容量只有 4K,在其常驻区内有一加载程序,能将主存贮器中的例行程序加载到此存贮器中去。在 ASC 系统中,各外围机只有公用的 4K 只读存贮器存贮那些较短的、被频繁使用的例行程序,而执行其它程序都需要访问主存贮器。

对共享主存的连接方式,根据各外围机是否具有自己的运算部件又可分为两类:

(1) 各外围机具有自己的运算部件,独立地与主存相连。中央处理机、各外围处理机与多体交叉的主存贮器之间可经图 4.6 的交叉开关相连, Burroughs B-6700 属于这种结构。

(2) 各外围机合用运算部件,并通过公用部件与主存相联。由于外围机的指令操作相当多的部分是属于访问主存或外围机存贮器,而运算部件的执行时间往往比存贮器周期短得多,从而为各外围机分时使用公用运算部件提供了可能性, CDC—CYBER 和 ASC 都属于这一类, ASC 的八个外围机分时公用一个运算部件、一个指令处理部件和一个 4K 容量的只读存贮器。这样,能够降低外围机的造价,但却使控制复杂化。

二、不共享主存的连接方式

STAR—100 是属于这种连接方式,它的 I/O 系统有若干个站,每个站包括站控制器和站缓冲器。站控制器内有站处理机、显示终端、一台小型磁鼓以及与站缓冲器和中央处理机通讯的接口;站缓冲器内有 32 K 16 位字长的存贮器和输入输出设备与缓冲器间的接口等。站控制器通过执行站处理机存贮器内的程序来控制站的操作。站处理机比共享主存方式中的外围机具有更大的独立性。

对输入操作,从输入设备输入的数据先放在站缓冲器的存贮器中,然后在站控制器的控制下,再从站缓冲器通过传送通路送往主存。

其实,这二种连接方式并没有本质的区别,主要区别在于后者的外围机附有更大容量的缓冲存贮器。

下面我们简介 CDC—CYBER 170 的外围处理机 I/O 系统。

5.2 CDC—CYBER 170 外围处理机 I/O 系统

5.2-1 概述

CYBER 170 系统计算机的基本结构示于图 4.33 中, 它包括 172、173、174、175 等各档, 它们的处理能力、数据存取速度和 I/O 配套能力不同, 但基本结构是相同的, 都包括有中央处理机、主存贮器、外围处理机 I/O 系统、I/O 通道转换器等。

CYBER—172 的外围处理机子系统 (PPS—Peripheral Processor Subsystem) 包括分时使用主存的 10 台外围处理机 (PPU—Peripheral Processor Unit), 共享 12 个输入输出 (I/O) 通道, 分时使用多台外部设备。每个 PPU 有一个 MOS 存贮器, 容量为 4096×13 位 (13 位中, 一位为奇偶位)。外围处理机 PP0 装有系统监督程序, PP1 装有操作台显示程序, 其余 8 个 PPU 各装有自己的常驻程序, 每台 PPU 都能独立工作, 执行有关的外围处理机程序, 管理外部设备。

各台 PPU 的指令系统都相同, 完成算术/逻辑运算、读/写中央存贮器、与外部设备交换信息等功能。10 个 PPU 总的信息流量最大为 $20 \times 10^6 \times 12$ 位/秒。

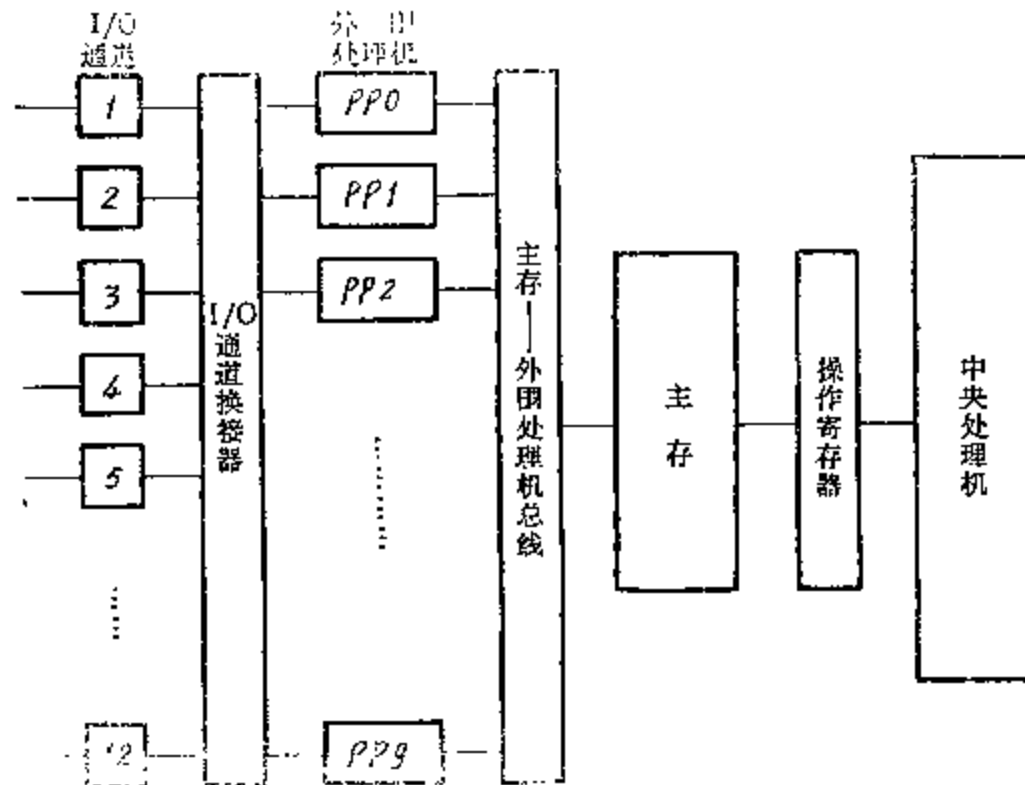


图 4.33 CYBER 170 系列结构

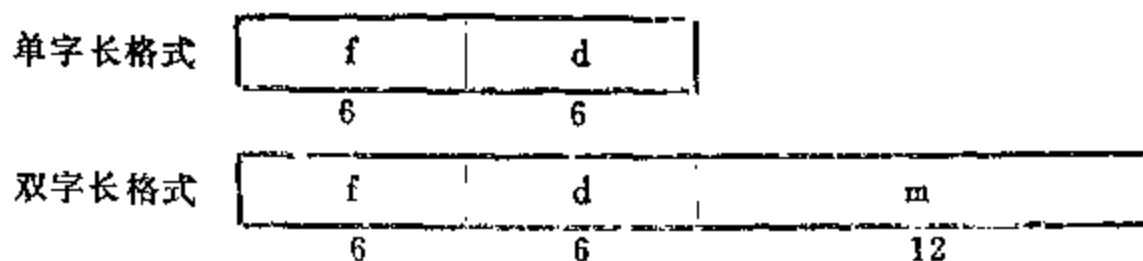
5.2-2 指令系统

外围处理机使用单字长 (12 位) 和双字长 (24 位) 两种指令格式, 如图 4.34 所示。两种格式都有着灵活的操作数编址方式, 有直接编址、相对编址和变址编址等。

外围处理机的指令系统包括: 算术型指令 17 条; 逻辑运算型 30 条; 读/写主存贮器型 4 条; I/O 型指令 12 条以及更换转移型指令 3 条。

用外围处理机指令编制的程序称为外围处理机程序, 存放在主存程序库中, 可随时调用来管理外部设备。

这里顺便提一句, CYBER170 系列与 IBM370 系列不一样, 它的中央处理机不能与外部设备交往, 因为在 CPU 指令系统中没有输入/输出指令, 只有外围处理机才能与外部设备



f: 操作码

d: 地址、转移参数、通道号

m: 转移地址、操作码、外围处理机存储器的开始地址

图 4.34 CYBER170 系列外围机指令格式

通讯。当用户程序需要输入/输出时，中央处理机要做的唯一工作是请求调用外围处理机；在发出请求后，中央处理机可以马上继续执行用户程序，避免了象 IBM370 那样的前处理，使中央处理机能专用于用户程序的执行。

下面选择介绍几条和访主存、与 CPU 通讯及控制 I/O 通道有关的外围处理机指令。

(1) 更换转移

其功能是把外围机 A 寄存器 (18 位) 的内容送给 CPU，作为 CPU 更换转移的主存起始地址。即 CPU 的现场 (24 个操作寄存器: 8 个操作数寄存器 $X_0 \sim X_7$ 、8 个地址寄存器 $A_0 \sim A_7$ 、8 个变址寄存器 $B_0 \sim B_7$ ；指令寄存器和 6 个辅助寄存器) 与主存中由 A 地址开始的 16 个字 (相当于新现场) 进行交换，随后，CPU 按新的现场继续执行。

这条指令相当于执行对中央处理机的中断，通过交换新、旧现场 (类似于 IBM370 的交换新旧 PSW) 使 CPU 转入由外围处理机要求的处理程序。类似的还有监督更换转移指令，除执行上述动作外还置“1”监控标志触发器，使 CPU 处于监控态 (相当于 IBM370 的管态)。

(2) 四条读/写中央存储器指令——60d、61md、62d、63md

每台外围处理机使用这四条指令可与主存进行单个字或成块字的读、写。

60d——其功能是把 PPU 的 A 寄存器所指明的主存单元的 60 位字送到外围处理机从单元 d 开始的 5 个单元。

61md——它是把主存内由 (A) 所指出的主存地址开始的一组字读到外围机存储器从地址 m 开始的单元中，从主存读出的字数由 (d) 指明。

62d 与 63md 两条指令则是从外围机存储器写入主存。除了数据流的方向相反外，执行的功能与 60d 和 61md 读指令相同。

(3) 从 64 到 77 为 12 条 I/O 指令，外围处理机使用这些指令控制和使用所有的外部设备和通道。例如 74d，这条指令为启动通道 d 指令，它启动 d 所指定的通道，建立通道 d 工作标志，发一个“工作”信号沿该通道送到所连接的输入输出设备。又如 73md 指令，它是向通道 d 输出从 m 开始的 (A) 个字。

5.2-3 CYBER170 外围处理机的特点

如图 4.35 所示，所有 10 台外围处理机分时使用同一个算术/逻辑部件，我们在这一节主要就是介绍这个特点。

每个 PPU 都有四个寄存器：A (18 位) 寄存器是一个多用途的累加寄存器，它保存一个算术/逻辑运算数、或输入/输出操作数，也可以是主存地址、I/O 数据交换的字数，P 寄

寄存器(12位)是程序地址寄存器; Q寄存器(12位)是多功能寄存器,它保存直接或间接地址、通道号以及转移计数值等; K寄存器(9位)则是保存指令操作码 f(6位)和指令执行大周期数(3位)。

一、“筒”与“槽”

由图 4.35 看出, 10 个外围处理机的 A、P、Q、K 寄存器, 首尾连接组成了 180 位、120 位、120 位、90 位的移位寄存器, 形成一个环称之为“筒”, 10 个外围处理机依次排列在筒壁上, 筒的缺口处称之为“槽”, “槽”位于筒的第 9 列与第 0 列之间, 公共运算部件就在槽内。筒壁内各个 PPU 的寄存器移入槽时, 执行指令的一步功能, 或向外围处理机存贮器、或向主存、或与通道交换一次信息。出槽时进入第 0 列, 而后经过一个大周期再次入槽。因此, 一条外围处理机指令可能要经过许多个大周期才能完成。

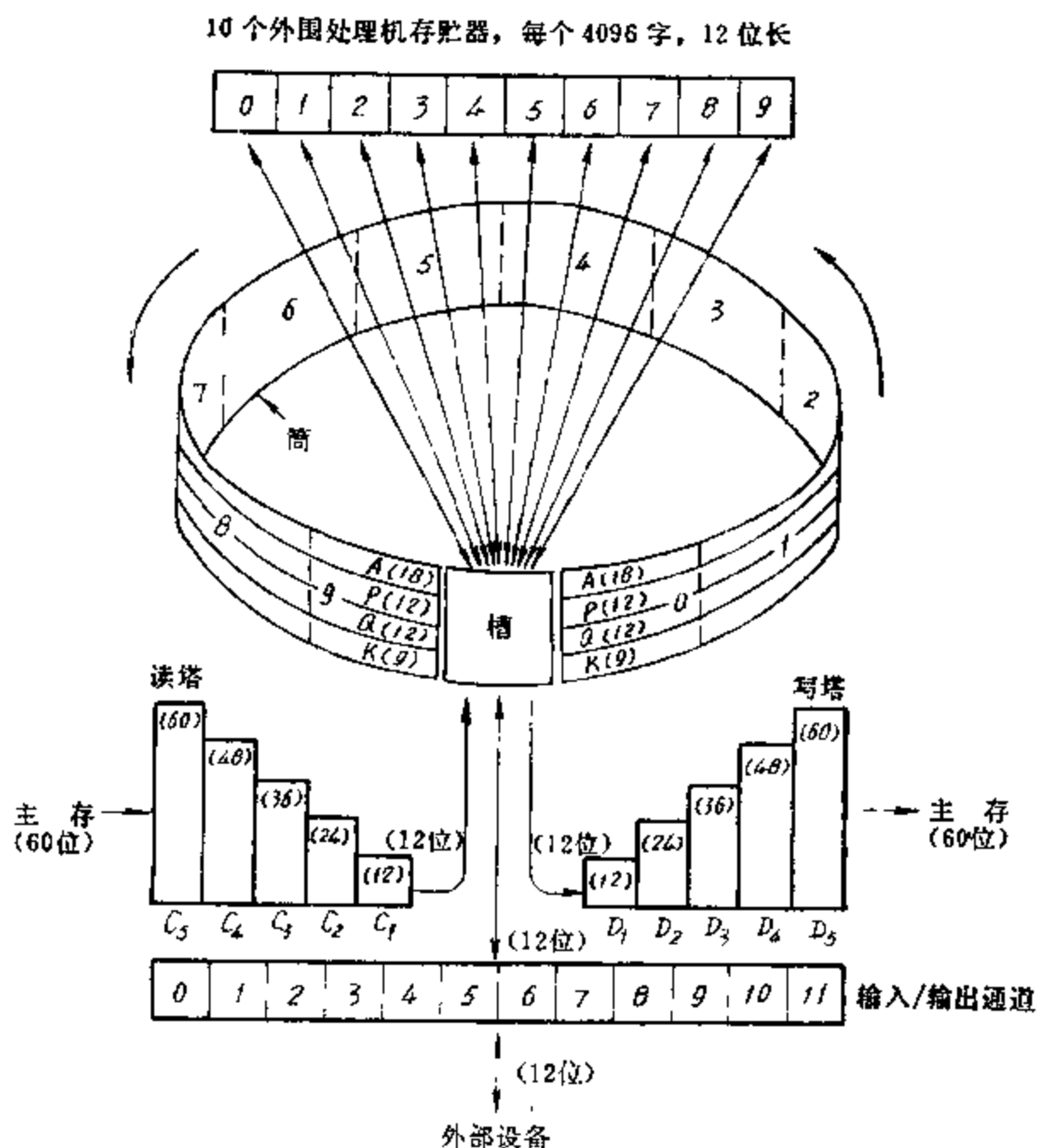


图 4.35 外围处理机子系统示意图

二、读/写塔及其控制

由于主存的字长是 60 位, 而外围处理机存贮器的字长是 12 位, 因此在它们之间交换信息时, 需进行拆卸和装配, 为此在槽内中设置了“读塔”和“写塔”。

由于“读塔”和“写塔”也是 10 个外围处理机分时公用的, 而且在一个大周期中可能有几个外围处理机与主存交换信息, 因此还必须解决好塔的控制问题, 保证外围处理机按正确顺序存取数据, 不致使塔中的数据发生错乱。

(1) “读塔”

假定当前刚入槽的外围处理机为 PP8，它执行的指令是从主存的 (A_8) 单元读出一个字，这需分 5 次写到外围机存储器从地址 d 到 $d+4$ 的 5 个单元中。当 PP8 入槽后，它首先检查在此刻之前有无别的外围机已向主存发了请求信号；若无，则通过优先判别电路向主存发出读请求信号，同时将 A_8 内容送主存地址寄存器。当该读请求信号被主存接收，则在 PP8 下次入槽前，由主存读出的字就已进入“读塔”的 C_5 寄存器 (60 位)。在 PP8 再次入槽时，将 C_5 的高 12 位送至外围机存储器数码寄存器 Y，再写入外围机存储器 d 单元，同时将 C_5 移至 C_4 (腾出 C_5 为别的外围机用)，地址加“1” (即 $d+1 \rightarrow d$)。又经一个大周期，在 PP8 又再次入槽时，则将 C_4 的高 12 位送至外围机存储器的 $d+1$ 单元， C_4 的低 36 位移至 C_3 ，地址再加“1”；依此，PP8 由主存读出的一个字，需用 5 个大周期，经由 C_5 到 C_1 ，才能送入外围机存储器。 $C_1 \sim C_5$ 寄存器的使用采用了流水的概念。

在一个大周期内，“读塔”只能响应一个外围处理机从主存读出信息到 C_5 的请求，如果出现有两个外围处理机在一个大周期内请求从主存读出，则另一个必须等待到下一个大周期，待 C_5 寄存器空出后才能请求从主存读出。另外，“读塔”在满负荷的情况下， C_1 到 C_5 中分别存放着按顺序入槽的外围处理机所需的数据。

设 PP0、PP1、PP2、PP3、PP4 五个外围处理机在同一周期同时开始执行读主存的指令，其发访主存请求和塔中数据流的情况如下表所示。 T_i 表示第 i 个大周期。从表中可看出， T_6 时 PP0 虽然执行 $C_1 \rightarrow Y$ ，但由于此时 C_5 还是满的，所以 PP0 要等到 T_7 时才能发请求。

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
PP0	发请求	$C_5 \rightarrow C_4$	$C_4 \rightarrow C_3$	$C_3 \rightarrow C_2$	$C_2 \rightarrow C_1$	$C_1 \rightarrow Y$	发请求	$C_5 \rightarrow C_4$
PP1		发请求	$C_5 \rightarrow C_4$	$C_4 \rightarrow C_3$	$C_3 \rightarrow C_2$	$C_2 \rightarrow C_1$	$C_1 \rightarrow Y$	发请求
PP2			发请求	$C_5 \rightarrow C_4$	$C_4 \rightarrow C_3$	$C_3 \rightarrow C_2$	$C_2 \rightarrow C_1$	$C_1 \rightarrow Y$
PP3				发请求	$C_5 \rightarrow C_4$	$C_4 \rightarrow C_3$	$C_3 \rightarrow C_2$	$C_2 \rightarrow C_1$
PP4					发请求	$C_5 \rightarrow C_4$	$C_4 \rightarrow C_3$	$C_3 \rightarrow C_2$

(2) “写塔”

这是为向主存写数而设立的部件。“写塔”亦有 $D_1 \sim D_5$ 五级寄存器。从外围处理机存储器读出的第一个字送到 D_1 ，第二个字送到 D_2 的低 12 位， D_1 则移到 D_2 的高 12 位。依次类推， $D_2 \rightarrow D_3$ ， $D_3 \rightarrow D_4$ ， $D_4 \rightarrow D_5$ ；需经 5 个大周期才在 D_5 中形成要写入主存的字。在 D_4 进入 D_5 时向主存发写数请求，待回答后把 D_5 中的数写入主存。

三、I/O 通道

这里所指的“通道”和上一节所讲的“通道”不是一个概念，CYBER 的 I/O 通道只起通路连接作用。外围处理机可与 12 个通道中的任一个相接。每条通道可接 8 台外部设备，用设备号来选择。每条通道都有一个双向 13 位 (一位奇偶位) 的通道寄存器，一个工作/不工作和一个满/空状态标志触发器，利用这两个通道标志控制外围处理机与外部设备的通讯。

利用前述外围机的 12 条 I/O 指令控制通道与外部设备交换信息。

下面阐述外围处理机是怎样与外部设备进行数据交换的。

1. 输入

从外部设备向外围处理机输入数据，可划分为如下几个阶段：

(1) 选择设备阶段

首先，外围处理机在其对应列的 A 寄存器中放置所选设备的设备号，并将“满”、“工作”状态标志触发器置“1”。接着，使用外围处理机指令把 A 寄存器中的内容送往通道，用以连接外部设备。外部设备接到信号后，给外围处理机回送一个“不工作”信号，清除通道的“满”、“工作”标志触发器，设备选择阶段结束。

(2) 命令发送阶段

外围处理机再在 A 寄存器中放置确定信息交换的性质和格式的命令，并将它送至外设控制器，然后再将数据交换个数存放到 A 寄存器中。接着通过外围机指令启动通道。

(3) 传送数据阶段

执行外围机输入指令。每当外部设备给外围处理机发送一个数据，就置“1”“满”触发器。在数据送入外围机存贮器单元后，清除“满”触发器，给外部设备送一个“空”信号，并做 $(A)-1 \rightarrow A$ ，使外部设备可以继续送数。重复此过程，直至 $A=0$ ，传送结束，断开通道。

2. 输出

外围处理机向外部设备传送数据时，其连接设备，发送命令，取状态等准备工作与输入相同，只是执行的是输出指令。每当外围处理机向输入/输出缓冲寄存器放置一个数据时，就置“1”“满”触发器，给外部设备送一个“满”信号。当外部设备收到数据后，就接收此数据，回送一个“空”信号给外围处理机，清除通道“满”触发器。如此往复，直至 $A=0$ ，交换结束，断开通道。

5.3 通道结构 I/O 系统与外围处理机结构 I/O 系统的简单比较

从上面的讨论可以看出，对于与 I/O 设备的交换信息而言，具有外围处理机结构的 I/O 系统与通道结构的 I/O 系统没有本质差别。只是在数据块的传送中，后者是通过硬件的时序线路，通过执行通道命令来实现的；而前者则是通过执行若干条外围机指令来实现的。后者输入的数据是直接送入主存，前者则是送入外围处理机的存贮器。

显然，通道结构的 I/O 系统，由于外部设备是被接入固定的通道，不能用程序来改变其连接位置，所以，灵活性差；而外围机结构的 I/O 系统，则可按程序需要，动态地改变 I/O 设备与外围机的联接，灵活性好。采用固定通道，它可以根据各种输入输出设备各自传送速率的不同，设计多种通道，以防止低速输入输出设备占用高速通道。但外围处理机结构的 I/O 系统也不必使每个外围通道按高速设备设计，因为可以用多个通道合并来处理高速设备的数据传送。由于外围通道这种任务可随机变化的特点，因此人们有时称它为“浮动通道”。

另外，通道结构的 I/O 处理机（即通道）的专用性比外围处理机强，因而它的结构可以更加简化，利用率可以更高。

然而，它们之间的主要区别在于前面 § 5.1 讲的那四点。

尽量使中央处理机摆脱 I/O 管理的负担,对于发挥中央处理机的高速效能很有意义。随着硬件价格的下降,处理机会更多地采用上一章讲过的重迭、流水技术;然而,前面讲过,采用这些技术的处理机一旦遇上“中断”,其高速性就发挥不了。所以,通道型结构 I/O 系统那种每调用一次 I/O 设备,就得经“访管”指令进入中断,再进入 I/O 管理程序的做法,从 CPU 资源来看是不合理的。I/O 管理程序的功能简单,它们是不需要,也往往是用不上 CPU 的高速性能,甚至浮点运算也用不上;对下一章要讲的高速缓冲存储器 Cache 也是如此。因此,当运行 I/O 管理程序时,CPU 各部件的利用率就显著下降。对于 § 5.1 所讲的第 3 点的那些运算和操作,也很难用得上 CPU 所具有的高速性。从这些看,尽可能把与输入、输出有关的操作及运算从 CPU 挪出去,移给外围机,无疑是更加合理的。

我们在第一、二章都已讲过,操作系统的分布(分散)实现比集中实现合理。外围处理机不只是能把设备管理和文件管理的主要操作接过去,而且对作业调度也可做很多工作,这样,不仅可实现用户程序和操作系统的并行工作,还可实现操作系统各部分的并行工作。

还有,随着硬件价格的下降,尤其是微处理器的发展,外围处理机的价格必然会显著下降,其结构也会规整化(例如,CYBER 那种分时使用运算部件的做法就没有必要了),其功能也会进一步加强。现在,不只是巨、大型机采用了外围机,就是中、小型机,如 HP-3000,也采用了外围处理机的思路,在外设控制器和子通道内增设了微处理器。这样,I/O 管理中的很多工作,错误检测,对外设中断请求的管理等都由这些微处理器完成,使得 CPU 的 I/O 管理既简单,又统一。另外,由于各个外设控制器配有微处理器及相应的缓冲存储器,就使得各个外设之间(如行式打印机和磁盘之间)能不经主存直接交换信息。

总之,增强 I/O 处理机的功能,向分布处理方向发展是今后的必然趋势。

主要参考文献

- [1] D. J. Kuck, "The Structure of Computers and Computations" Vol. 1, John Wiley & Sons, 1978.
- [2] H. S. Stone, "Introduction," Ch. 1 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [3] J. P. Hayes, "Computer Architecture and Organization," Mc Graw-Hill, 1978.
- [4] IBM 4300 Processors Principles of Operation for ECPS, VSE Mode, GA22-7070-0, 1979.
- [5] I. Flores, "Peripheral Devices", Prentice-Hall, 1973.
- [6] A. S. Tanenbaum, "Structured Computer Organization," Prentice-Hall, 1976.
- [7] K. B. Maglely, "Introduction to Minicomputers," ch. 4 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [8] W. G. Lane, "Input/Output Processing," Ch. 6 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [9] R. L. Sites, "Operating Systems and Computer Architecture," Ch. 12 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.

- [10] DJS-200 系列机通道, 1975.
- [11] D. J. Kuck(ed.), "High Speed Computer and Algorithm Organization," Academic Press, 1977.
- [12] 赵厚生, "输入、输出系统," 华东师范大学, 1980
- [13] A. C. Shaw, "The Logical Design of Operating Systems," Prentice-Hall, 1974.
- [14] J. E. Thornton, "Design of a Computer, the Control Data 6600," Scott Foresman, 1970.
- [15] S. E. Madnick and J. J. Donovan, "Operating Systems," Mc Graw-Hill, 1974.
- [16] K. J. Thurber, et al, "A Systematic Approach to the Design of Digital Bussing Structures," AFIPS Conf. Proc., Vol. 41, 1972, pp.399-420.
- [17] G. Morrow and H. Fullmer, "Proposed Standard for the s-100 Bus," Computer, Vol. 11, No.5, May 1978, pp. 84-89.
- [18] "CDC CYBER172 外围处理机子系统," 电子计算机动态, 1978, 9.

第五章 存贮体系

§ 1 引 言

1.1 存贮技术：容量、速度与价格的矛盾

大家知道，存贮器系统主要是由存放程序 and 数据的存贮器硬设备 and 控制 or 管理该存贮信息的算法（包括用软的 or 硬的方法实现）构成。虽然存贮器的大小、特性和物理构成，三十年来有了显著变化，然而基本的要求仍然是：大容量、高速度和低价格。存贮体系就是为了满足这些要求而发展起来的。

1.1-1 存贮器的基本要求

容量、速度和价格是评价存贮器性能的主要依据。

(1) 容量

用 W 表示存贮器的字长（用位数或字节数表示）， n 表示每个存贮器的字数， m 表示并行工作的存贮器个数，则

$$\text{总存贮器容量 } S = W \cdot n \cdot m$$

(2) 速度

用访问时间 T_A 来衡量存贮器的访问速度。定义 T_A 是存贮器从接收读申请到读出信息送到存贮器输出端的这段时间，它取决于存贮介质的物理特性和访问机构的类型。对于破坏读出及动态存贮器，只有重写或再生完成后，才能进行新的访存，就是对于不破坏读出的静态存贮器，由于读放、驱动源和存贮体走线都需有一段稳定恢复时间，所以在读出后也不能立即进行访问；因此除 T_A 外，还得定义存贮周期 T_M ，它是处理机可以连续二次启动该存贮器所需的最小时间间隔，显然， T_M 比 T_A 大，二者均是衡量存贮器性能所必需的。 T_A 在确定处理机与存贮器的时间关系中是重要的，因为它是处理机在启动一个访存申请后必须等待的时间。在 T_M 的余下时间，处理机和存贮器可以同时进行各自的操作。除 T_M 外，还要定义 B_m ，它是存贮器被连续访问时，可以提供的数据传送速率，也称频宽，通常用每秒钟传送信息的位数（或字节数）来衡量。影响存贮器频宽的一个因素是存贮器总线宽度 w ，它是总线可以同时传送的位数（或字节数）。 w 一般是，但并非一定是与标准的存贮器字长 W 一致；若不一致时，则单位时间一个存贮器能为处理机提供的信息量就不是

$$B_m = \frac{W}{T_M}, \quad \text{而是} \quad B_m = \frac{w}{T_M} \quad (\text{位/秒})。$$

若有 m 个存贮器并行为处理机提供信息，则可能达到的总的存贮器频宽为 $\frac{w \cdot m}{T_M}$ 。

(3) 价格

设 C 是具有 S 位存贮容量的存贮器的总价格，则每位价格为 $c = \frac{C}{S}$ 。

它不仅包含了存贮单元本身的价格，也包含了为该存贮器操作所必不可少的外围和访问

电路的价格。

$$\text{存贮器总的价格 } C \text{ 正比于 } \frac{S}{T_M(\text{或 } T_A)} = \frac{W \cdot n \cdot m}{T_M(\text{或 } T_A)}。$$

此外，影响存贮器性能的其他因素还有：访问方式（如随机的还是顺序的），信息的可更换性（如只读的、可读可写的、程序可编只读的），存贮物理介质（如电子的、磁的、机械的），存贮的永久性（如破坏性读出、动态存贮、易失性）等等。这里，易失性是指的所存信息在电源去掉后易于丢失的性质。

设计存贮系统的主要目标是：在尽可能低的成本下，提供尽可能高的速度及尽可能大的存贮容量。

1.1-2 主存和辅存

在计算机发展初期就已经认识到：由于能与处理机速度相配的高速存贮器的价格太高，因此总容量要求很大的存贮系统，采用仅仅一种工艺的单一存贮器是行不通的；机器内必须有多种存贮器，使所存的信息以各种方式分布在物理特性不同的各种存贮部件上。它至少需要有两种：主存和辅存。

大家知道，整个的程序和其数据可以看成是这样一个空间，从每个时间来看总可分成三个部分：“活跃的”部分表示当时正在被 CPU 使用的部分；“静止或安静的”部分表示曾经用过，但此时不在使用的部分；“待命的”部分表示至今未用过，或者只是当出现某些例外情况，例如诊断等，才被采用的部分。由于系统程序和应用程序所需要的总存贮空间一般超过主存容量，因此只能是把当时“活跃的”部分放在快速，每位价格较高、容量较小的主存中，他们能被 CPU 直接访问；而把其余部分放在每位价格要低得多的低速、大容量的辅存中作为后援；由存贮分配来决定每个时刻信息在各级存贮器中如何分布，它不断地把新的“活跃的”部分调入主存，而把其余部分仍放在或放回辅存。

在七十年代，具有代表性的存贮技术的主要性能如表 5.1 所示（表中所列数据是指的数

表 5.1 七十年代主要的存贮技术性能

工 艺		价格 c \$/位	访问时间 T_A 秒	访问方式	可更换性	永 久 性	存贮 介质
主 存 技 术	双极型半导体	10^{-1}	10^{-8}	随 机	读/写	不破坏读出， 易失性	电子的
	金属氧化物半导体 (MOS)	10^{-2}	10^{-7}	随 机	读/写	破坏或不破坏 读出，易失性	电子的
	磁 心	10^{-2}	10^{-6}	随 机	读/写	破坏读出， 非易失性	磁
辅 存 技 术	磁盘 (磁鼓)	10^{-4}	10^{-2}	随机或半随机	读/写	不破坏读出 非易失性	磁
	磁 带	10^{-5}	10^{-1}	顺 序	读/写	不破坏读出 非易失性	磁
	穿孔卡和纸带	10^{-6}	10	顺 序	只 读	不破坏读出 非易失性	机械

量级而不是准确值)。

在这些存贮技术中,对改善系统性能价格比起主要作用的是MOS 随机存贮器和磁盘。在国外, MOS 存贮器在速度、体积、每位价格和所能构成的容量上,于七十年代初就已超过了磁心存贮器,目前的主存几乎都是采用 MOS 存贮器。参看图 5.1(a),动态 MOS随机存贮片子容量在 1971 年初为 1K 位,而后以每二至三年增长四倍的速率发展。1980 年 16K 位片子已经大量生产,而 1977 年夏季首次报导的 64K 位片子在 80 年也已投入市场。按照这种趋势发展, 4M 位片子在九十年代就可出现。增大片子的面积直至例如直径为 5 吋,对增大 MOS 存贮器单片容量也会起重要作用,但需要进一步采取措施解决合格率问题。高速双极型存贮器多被用在高速缓冲存贮器 (Cache) 中。

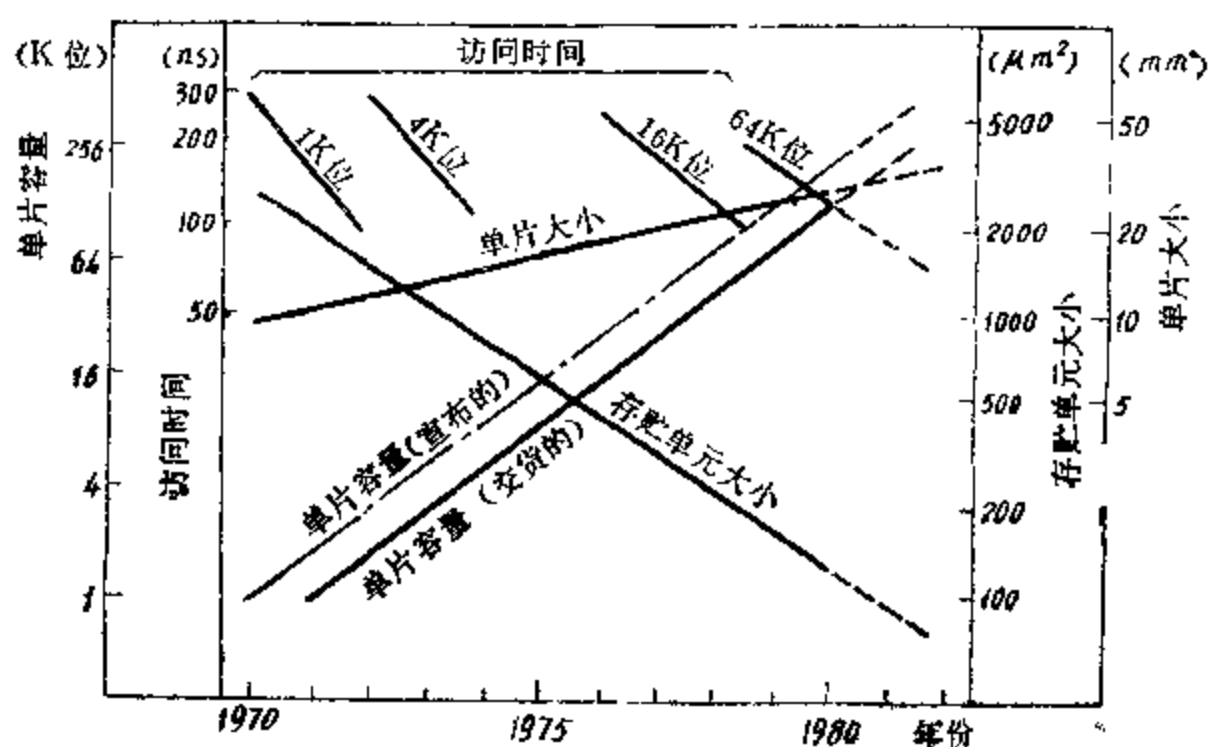
辅存相对主存来说,速度慢、容量大而每位价格低,通常存放当前不活跃的程序和数据、系统程序或大的数据文件等,一般不直接被处理机所访问,目前作为辅存的主要有磁盘(定头和动头)、磁鼓和磁带等。它们的访问速度主要受限于机械运动。以动头磁盘为例,为访问某个磁道上的某个信息块,所需的访问定位时间包括磁头臂径向移动到所需磁道的查找时间 t_{seek} 和该信息块首旋转移动到磁头下,开始读出所需的旋转时间或等待时间 t_{rot} 这两部分;它们都受限于机械运动。之后,数据就以同一速率向外传送,这个速率取决于盘片转速和信息的存贮密度。

表 5.2 典型磁盘的基本参数

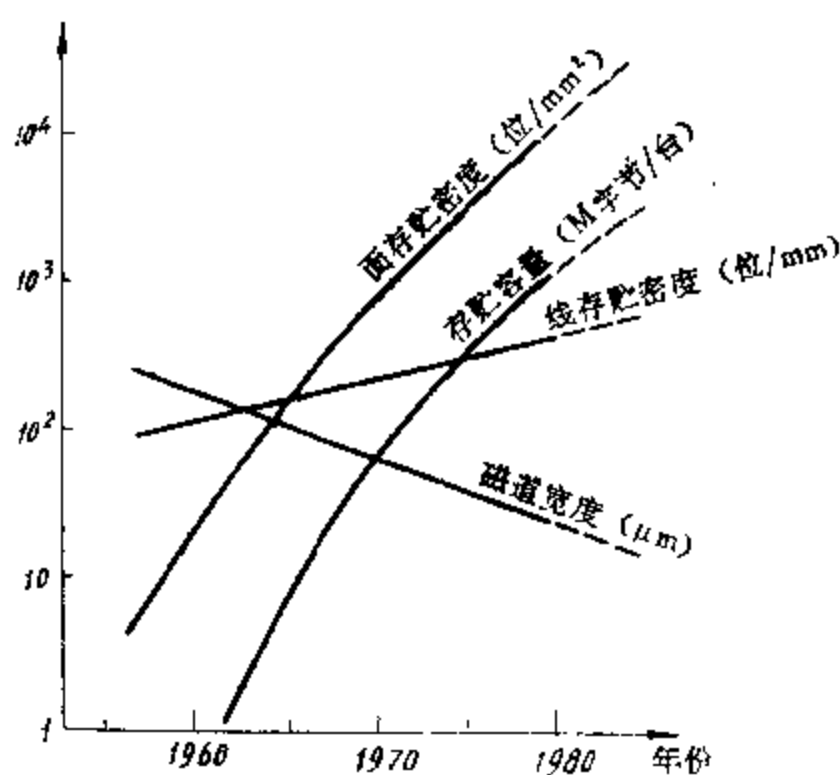
年 份	型 号	容 量 (兆字节)	面 密 度 (位/平方毫米)	访问 4K 容量信息 块的时间 (毫秒)
1956	IBM 350	5	3.1	925
1964	IBM 2311	7.3	170	113
1965	IBM 2314	29.1	340	85
1970	IBM 3330-1	100	1200	43.3
1973	IBM 3330-11	200	2320	43.3
1973	IBM 3340	70	2550	39.6
1975	IBM 3350	317	4740	39.6
1978	STC 8650	635	9456	

参看图 5.1 (b), 磁盘性能的改进主要是由于存贮密度的提高。磁盘的面密度从 1970 到 1980 年已提高了十倍。典型磁盘的基本参数如表 5.2 所示。预计线密度 1000 位/毫米, 道密度 50 道/毫米和面密度 5×10^4 位/毫米² 的磁盘不久就能投入实际使用,其容量可达 4×10^9 字节。可以肯定,至少在八十年代,磁盘仍将在辅存中占据重要地位。

早期, CPU 与主存在速度上是相配的。例如 1955 年第一台大型(就当时来讲)科学计算用机器 IBM704, 机器(CPU)周期(即拍宽)为 12 微秒,主存周期也为 12 微秒。但是,随着器件工艺的进展,如图 5.2(a)所示,虽然主存速度在二十年内提高了 2 个数量级,然而同期内,最快的 CPU 的拍宽已从几十微秒缩短到几十毫微秒,约 3 个数量级。就是说,到了七十年代,在合理的成本下,足够容量的主存,其存贮周期是比 CPU 拍宽大一个数量级。



(a) 动态MOS RAM 片子的发展

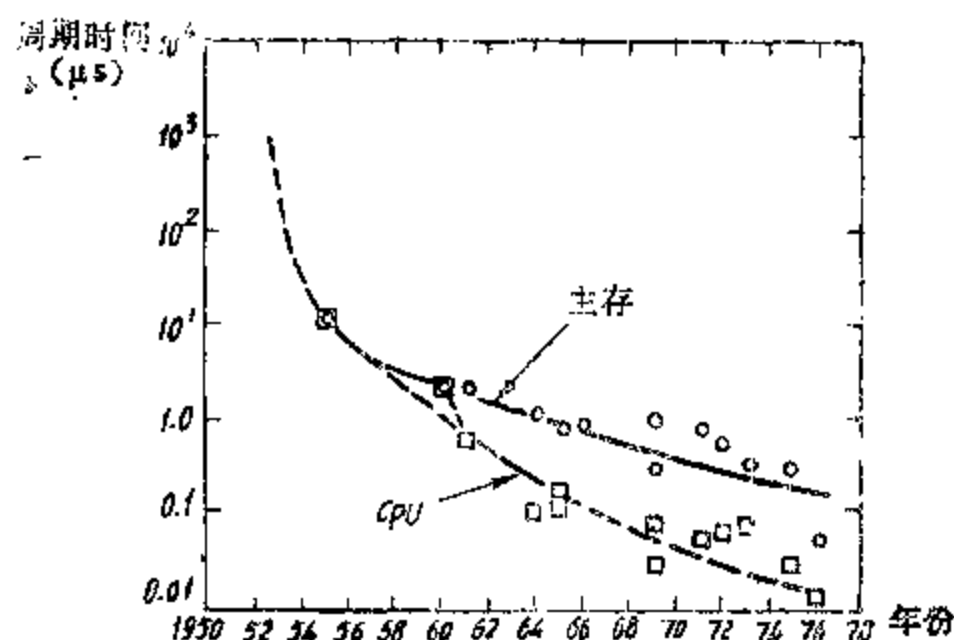


(b) 磁盘存储器的性能发展

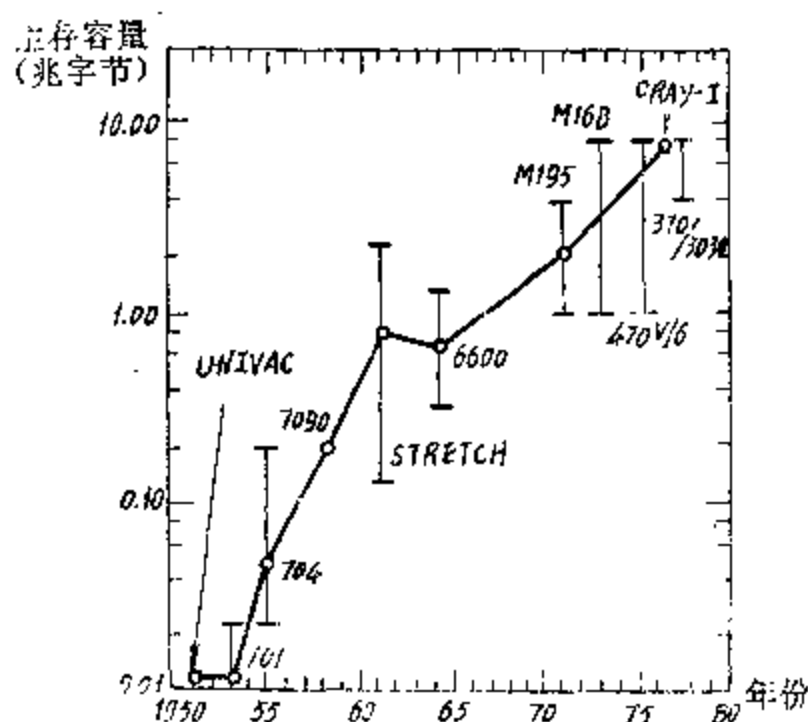
图 5.1 主要存储技术的性能发展

显然，必须从系统结构上采取措施来解决这个速度差，否则，CPU 的高速效能是发挥不了的。

从容量上看，尽管主存容量是在相当快速地扩大，如图5.2(b)所示，从1951年UNIVAC I 的 1K 字 × 91 位提高到 1976 年 CRAY-I 的 1M 字 × 64 位（相当于 8M 字节），1978 年的 IBM3033 的 8M 字节，目前，就是微型机中，其主存容量达 32K、64K 字节的已很普遍，而 1975 年问世的小型机中的高档 PDP-11/70，其主存已可达到 4 兆字节（2 兆字）。然而，由于系统软件和应用软件的迅速发展，主存仍然不可能存得下整个系统程序和用户程序，而只能存得下必须驻留在主存的操作系统，按需要临时调进主存的操作系统和当时正使用的编译程序以及多道用户程序中的“活跃”部分等。由于主存容量的增大总是落后于程序的不增大，就是在今后，当主存容量可达几百兆字节时，情况也仍然会是如此；因此，必须同时具备主存和辅存的状态定会长期存在。



(a) 主存周期 T_M 与 CPU 周期 T_{CPU} 随年份的变化



(b) 主存容量随年份的增加

图 5.2 主存速度和容量的发展

1.1-3 价格问题

现在，我们从容量、速度和价格之间的矛盾对主、辅存进行分析。人们总是希望主存的容量足够大以使处理机不必因为需经常访问较慢的辅存而耗费过多的等待时间，同时，也希望主存有足够快的速度。就是说，希望 W 、 n 、 m 大而 T_M 小。但是，价格 C 正比于 $M \cdot n \cdot m / T_M$ ，这样，速度越高，容量越大，则价格越高，它们之间是互相矛盾的。不仅是各种主存技术，就是各种辅存技术也是如此。存贮技术的多样性以至存贮体系的形成，主要原因就在于此。就一种存贮器来看，要求访问速度快只能以高的成本为代价，这是一种内部的固有特性。因此，就需要在价格和速度、容量之间进行折衷。其中， n 和 T_M 主要与器件工艺有关，而 W 和 m 则是由系统结构设计者确定。

我们在第一章 § 4.1 已经讲过，主存的每位价格随着存贮技术特别是半导体技术的发展，在不断下降，由图 5.3 还可进一步看清这点。通过近年来对器件工艺的研究和探索，预期本世纪还可降低几十倍甚至几百倍。每位价格如此大幅度降低，将使更大容量主存成为可

能。然而，主存的价格是等于每位价格乘以容量（总位数），它不能过高，不能在整机价格中占据过大的比例。

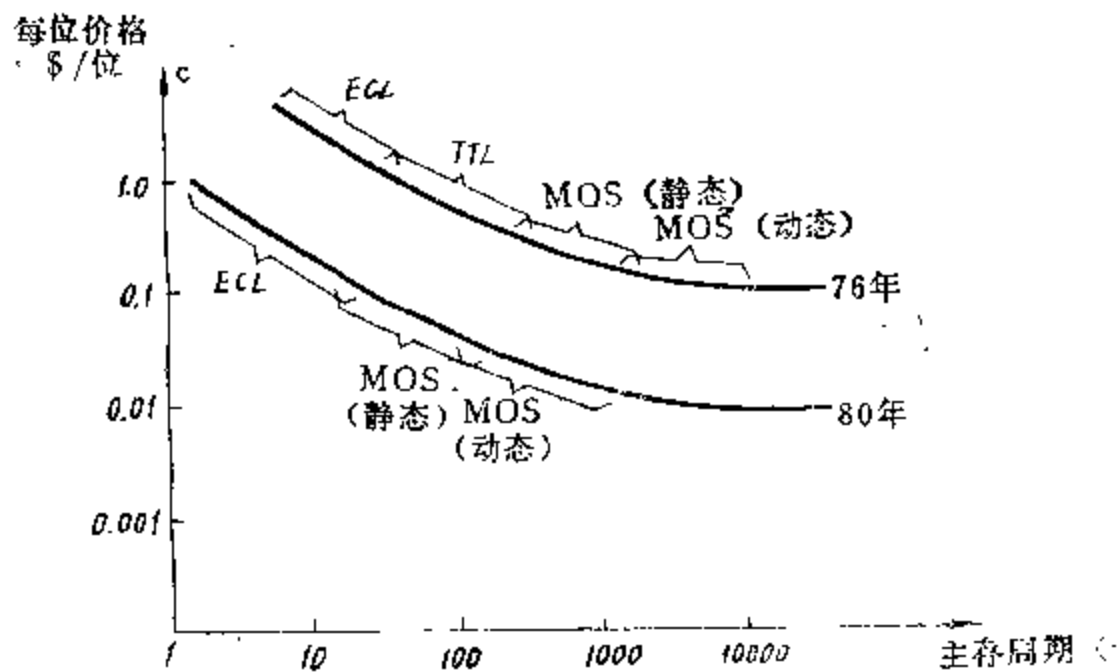


图 5.3 半导体存储器速度、每位价格随年代变化趋势

目前，在一般的计算机系统中，主存已成为最费钱的部分，相对处理机、辅存、I/O设备来说，主存的价格已占到整个系统硬件价格的大约 30%到 40%。从过去看，主存容量的扩大大致是正比于每位价格的下降，以使得主存总的价格基本保持不变。在可予见到的未来，主存的价格仍然是影响机器性能提高的重要因素。

相对主存来说，辅存的价格则便宜得多了，以 79 年磁盘的价格为例，容量为 67M字节的只是 1M字节主存的 30%~40%，而 176M字节的也才是 1M字节主存的 60%左右。虽然半导体存储器单片位密度几乎以 2 的方幂提高，但要求的辅存容量在急剧增大，要用半导体存储器片子来构成辅存，则仍然必须通过大量单片的互连来构成，其价格必然昂贵；因此，如前所述，磁盘和磁带在可予见的将来作为便宜的辅存仍然会继续在计算机系统中得到广泛应用。

把表 5.1 几种主要的主、辅存技术按访问时间 T_A 和每位价格的关系，可以用图 5.4 来描述。从图中可以看出，主存和辅存之间（即图中 MOS、磁心与磁盘之间）存在着一个明

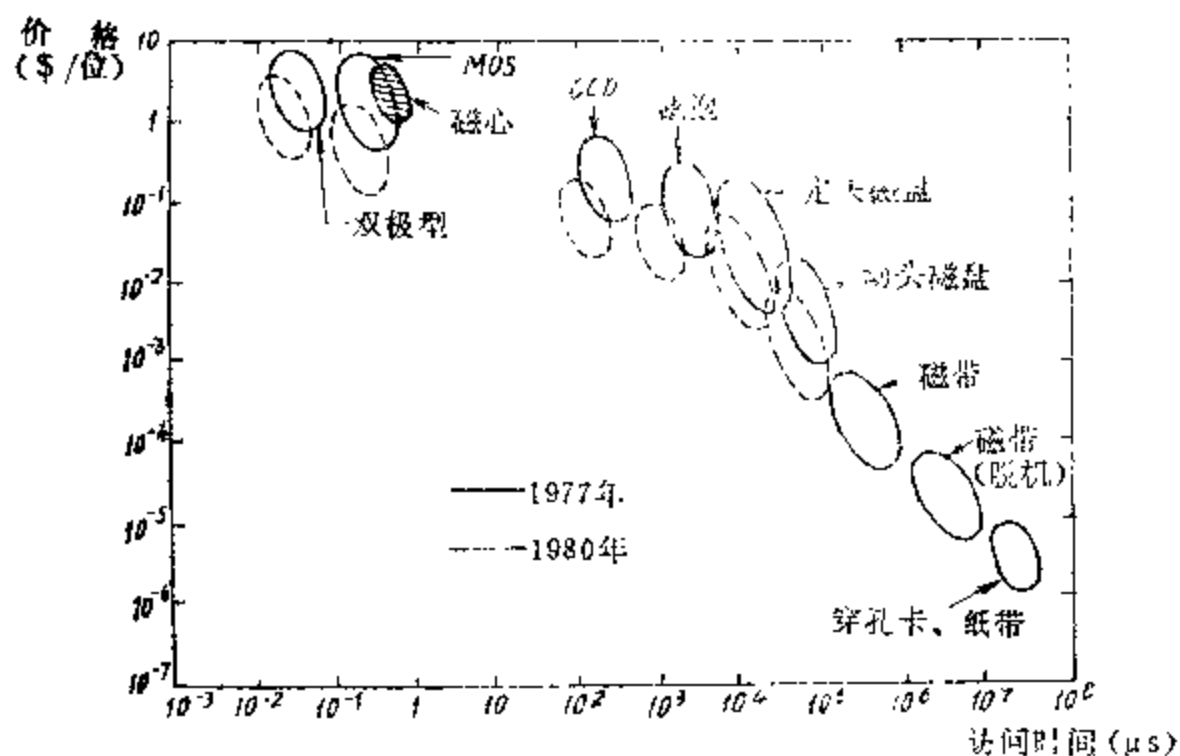


图 5.4 各种存储技术的访问时间与价格的关系

显的性能价格差距，这个差距对下面要讲到的存贮体系构成是不利的。另一方面，对于现有的存贮技术工艺方面的改进是设法不断减小每位价格 c 和缩短访问时间 T_A ，然而，随着时间的推移，各种技术和工艺在这两方面的改进速度大致相同，使得主、辅存在速度、价格上的这个差距依然存在。因此，为了能填补这种差距，以利于存贮体系的构成，器件厂正在研制所谓电子磁盘，用电子移位来代替慢速的机械运动。其中最有希望的是磁泡存贮器和电荷耦合器件存贮器，相比之下，磁泡更适宜大规模集成，同时又具有非易失性的优点而更有前途。虽然国外于 1978 年前后已推出 256K 位/片到 1M 位/片的磁泡存贮器样品，预计八十年代初可能出现每片几兆位的磁泡存贮器；但八十年代内电子磁盘仍然取代不了机械磁盘，因为在一个相当长的时期内，其每位价格仍然会比机械的昂贵；只有能把价格降低二个数量级并显著降低驱动功耗，这种替代才有可能。

1.1-4 主存容量、速度与 CPU 速度的关系

我们知道，CPU 连接到速度、容量和它不相配的主存上是不经济的。因为主存的存取速度过低或容量过小，都会使 CPU 大部分时间闲着，没有足够的数据和指令供它处理；而主存速度过高或容量过大时反过来会使主存经常闲着。因此，为了充分利用主存和 CPU，使之都能忙碌地工作，它们之间应保持信息流动的平衡。这是一个比较复杂的问题，它和机器的用途密切相关。我们先分析在 CPU 速度和主存速度之间所应有的比例关系。

一种看法认为，要达到平衡，CPU 的每拍应能取得一个操作数。然而由前面的图 5.2(a) 看出，这点是做不到的。其实，这个要求也是过高的，除了流水式处理机之外是不会每拍要用一个操作数的；而且，就是流水式处理机也不是每个操作数只用于一条指令的执行。因此，另一种看法，则是从分析每个操作数是被多少条指令的执行所用来寻找这种关系。这种分析虽然是很粗糙的，但可以用作根据 CPU 处理能力来大致估算出所需主存的容量和速度。这就引出了每个数据字节对应多少条指令执行的概念，称之为 I/B。

$$\frac{I}{B} = \frac{\text{为算某个程序所需执行的指令总数}}{\text{这个程序所处理的数据总字节数}} \quad (\text{指令数/字节})$$

B 中包括指令信息。由于循环、转移及其它操作，I 是远大于程序中指令的总数，可以差到二个数量级。I/B 与题目的类型及复杂性有关，而和 CPU 的结构关系较少。此比值大小已由很多人作了分析，对于不同类型的题目，I/B 的一般数值如表 5.3 所示，对于不同的大型机，此比值的差别是较小的。

表 5.3 对于不同类型题目处理每个数据字节所执行的指令数

题 目 类 型	大量作业上的 I/B 的平均值
科学计算类：FORTRAN 和其它	8
商业数据处理类：COBOL 和其它	4
交互式终端系统（例如管理系统）	1~2

科学计算类的 I/B 高于商业数据处理类的，这是由于在科学计算中，每个数据在程序执行时是被多次使用。例如，一组联立方程中的一个参数会在解题过程中多次使用，而商业数据处理中，对某个数据的操作简单，但数据量大。

由于一个程序，其执行的指令总数远比程序中指令的总数大得多，为简化讨论起见，可略去程序装入主存的时间。这样，当 I/B 确定后，因为 I 除以 CPU 的速度（每秒钟执行的指令数，IPS）就是解这题目所需的时间，在这段时间里共需由主存取得 B 个字节数据，所以主存的速度或周期可用下式估算出：

$$\text{主存周期(秒/字节)} = \frac{1}{\text{速度(字节/秒)}} = \frac{I/B}{\text{IPS}} \quad \begin{matrix} \text{(指令数/字节)} \\ \text{(指令数/秒)} \end{matrix}$$

至于主存容量和 CPU 速度之间应有什么样的比例关系才能保持平衡，这个问题更复杂。一种方法是经验法，就是对现有机器进行分析，找出比例关系。虽然大型机的主存容量迅速增大，如前面图 5.2(b) 所示，然而，如果把它们的主存容量（以兆字符为单位）与相应机器的 CPU 速度（以每秒执行百万条指令，MIPS 为单位）相比，则可得到图 5.5 的变化曲线。看出，从五十年代到七十年代，主存容量与 CPU 速度的比值大致是在 $0.5 \sim 1$ 之间波动。因此作为一个经验方法，我

们可以由

$$\frac{\text{M字符}}{\text{MIPS}} \approx 0.5 \sim 1,$$

估算主存容量。即大型机的主存容量可用下式估算：

$$\text{主存容量 (M字符)}$$

$$= \frac{\text{M字符}}{\text{MIPS}} \times \text{MIPS}$$

$$\approx (0.5 \sim 1) \times \text{MIPS}$$

显然，这种方法是认为已有的大型

机在主存容量和 CPU 速度之间基本相配。

这样，利用上述分析，对给定的 CPU 速度和 I/B 值，就可估出所需的主存容量和速度。例如，对 5MIPS 的 CPU，当典型题目的 $I/B = 4$ 时，设 1 个字符 = 1 个字节，则

$$\text{主存周期} = \frac{I/B}{\text{IPS}} = \frac{4}{5 \times 10^6} = 0.8 \text{ 微秒/字节},$$

$$\text{主存容量} = (0.5 \sim 1) \times \text{MIPS} = (0.5 \sim 1) \times 5 = 2.5 \sim 5 \text{ M字节}.$$

若典型题目的 $I/B = 1$ ，则所需主存速度要提高到

$$\text{主存周期} = \frac{1}{5 \times 10^6} = 0.2 \text{ 微秒/字节},$$

但容量不变，仍是 $2.5 \sim 5 \text{ M字节}$ 。这些值在 5MIPS 机器上还是比较典型的。

如果八十年代中期，一般大型机的速度能达到 80MIPS 的话，那么对 $I/B = 4$ 和 1 的情况，分别要求主存的容量和速度为：

$$\text{存贮容量} = (0.5 \sim 1) \times 80 = 40 \sim 80 \text{ M字节 (相当 } 10^8 \text{ 位)},$$

$$\text{存贮周期 (I/B = 4)} = \frac{4}{80 \times 10^6} = 50 \text{ 毫微秒/字节},$$

$$\text{存贮周期 (I/B = 1)} = \frac{1}{80 \times 10^6} = 12.5 \text{ 毫微秒/字节},$$

即主存周期比七十年代中、末期的值要提高约一个数量级。

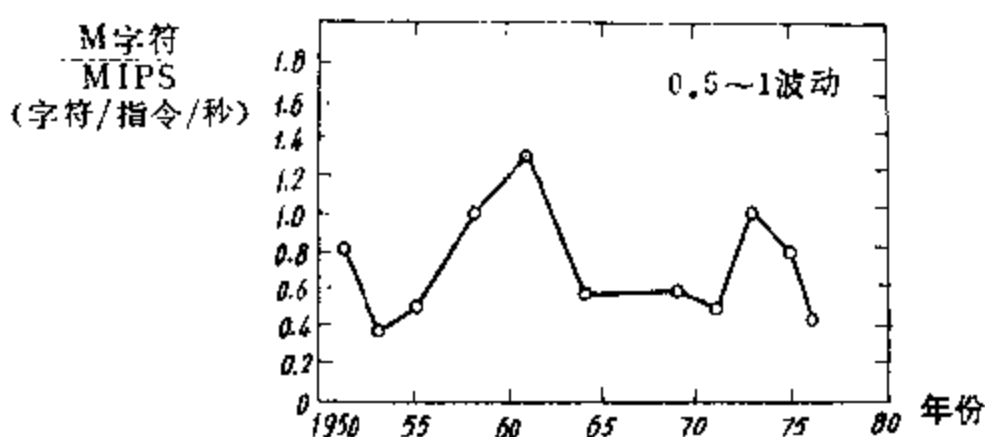


图 5.5 大型机主存容量与 MIPS 的比值随年份变化的情况（字符一般就是字节，但不一定是 8 位）

题目越复杂，要求主存容量愈大。在八十年代，随着计算机应用范围的扩大和深化，无论是事务处理（例如档案库、人口普查、文件管理、交通管理等），还是科学数据处理（如化学分析、天气预报、加速器设计、卫星照片传送处理、地质勘探等），都会要求有更大的主存容量。有人认为，主存容量可能发展到需要 $10^{12} \sim 10^{16}$ 位。

虽然计算机的总存贮容量可以通过磁盘、磁带、穿孔卡等后援接近于无限大，似乎任何大的题目都可以算，但实际上，由于这些存贮介质的低速，会使得很大题目的计算需要很长时间，以致长到（例如几年）实际上不可能算出。因此，机器所能解的题目的大小还是有限制的，它主要取决于 CPU 的速度、主存周期和辅存的数据传送速率。一个大的题目经常是分块地送入主存，而高的辅存数据传送速率可以提高 CPU 的利用率。下面要讲到的存贮体系将有助于简化大题目的处理。

综上所述，计算机系统和题目不断对存贮系统提出更高速和更大容量的要求，但它们受器件技术的限制又不可能同时都满足，而价格更是限制高速和大容量的重要因素。这就使得一方面要不断改进工艺，更新技术，另一方面要从系统结构上改进，采用存贮体系，软、硬结合，使存贮系统的性能价格比得到优化，以满足要求。

1.2 存贮体系原理

1.2-1 存贮体系的形成与发展

前面已经说过，由于在合理价格下，高速、大容量不能只用单一种器件来实现，因此，从一开始机器内部至少就有了二种存贮器。那么是否说，只要机器内有了多种存贮器就有了我们所要讲的存贮体系（也称为存贮层次，Memory Hierarchy）呢？不是的。我们认为，光有多种存贮器，如果它们之间不能形成有机的整体，那只能说是有了存贮器系统而并没有存贮层次。

让我们回忆一下，在早期的机器（参看图1.14）中，外（辅）存是作为外设的一部分，其编址与内存的编址无关，运算器在算题时一般是直接与内（主）存联系，外存的信息先要经过运算器送到内存后方能被运算器所使用。而且，程序如何一块、一块地由外存调入内存都得由程序设计者安排。这使得程序设计者必须化费很多时间和精力把主存放不下的（早期机器的主存很小，如 IBM650 才 1~2K）大程序预先分成有重迭的块，确定好这些程序块在辅存中的位置和装入主存的地址，而且在运算时，还要预先安排好各块如何和何时调入调出。所有这些，计算机系统均不能自动地进行。因此，主存与辅存并没有由系统结构和操作系统有机地联系在一起。这二者不能作为一个整体来看，从而也就没有存贮体系。这种情况往往会妨碍采用更好的算法，因为更好的算法经常要使程序变大并使分块复杂化，有时甚至会出现一个自然块就已超过主存容量的情况。

后来，随着通道与 I/O 处理机的出现，使得主存、外设、CPU 能同时工作，一块程序的执行与另一块程序的调入调出可同时进行。当然，如果主存中只有单道程序，新的一段程序调入时，老的一段程序已经执行完毕，从而只是通道在忙碌，CPU 却仍然在空等，这并没有能够实际地提高机器各部件的利用率。大家知道，在 CPU 和通道具有同时工作的能力时，还得采用多道程序，使这二者分别执行不属于同一道的程序，才能提高机器的解题能力（吞吐量），才能使得 CPU 运算和经通道的主、辅存信息交换实际上是可以同时地进行的。

虽然 CPU 的运算也要访问主存，然而利用主、辅存的数据速率差别大这个特点，当辅存与主存交换信息时，主存仍有很多空闲时间，这样 CPU 可插空访问主存。它们交替地使用主存，其时间关系如图 5.6 所示，即辅存和 CPU 是同时工作，但这二者是分时使用主存的。

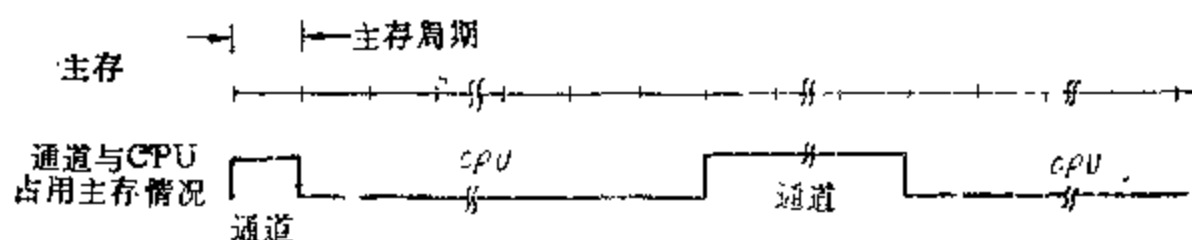


图 5.6 CPU 与通道分时使用主存

将以运算器为中心改成以主存为中心，主存必须与各个部件频繁地交往，加之目前主存要么读，要么写，同时只能访问一个单元，这就使得主存速度往往成为计算机系统效率进一步提高的瓶颈。

多道程序的发展，促进了管理程序而后是操作系统的形成和发展，使得程序设计者尽可能摆脱主、辅存之间的地址定位这一复杂工作，还逐步形成了支持这些功能的“辅助硬件”。在积累了程序分块的统计数据和经验的基础上逐步产生了这样一种概念和要求，就是如何从系统结构上通过软硬结合把主存和辅存统一成一个整体，如图 5.7 那样，使得从整体来看，

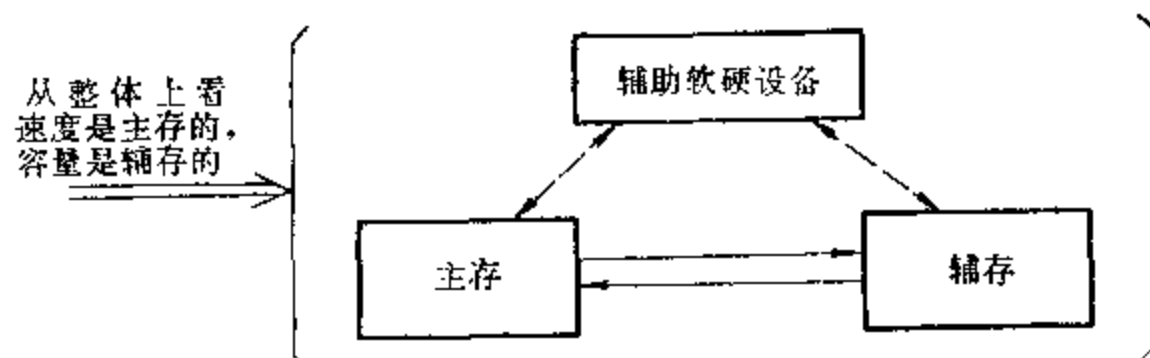


图 5.7 主—辅存贮层次

其速度接近于最快最贵的主存，但容量却是辅存的，而每位平均价格也接近于廉价慢速的辅存。我们认为，只有能够这样看时，才算形成了存贮体系（层次）。

存贮层次的形成对于提高存贮器系统的性能价格比很有好处，这是因为从速度上，主存的访问时间约为磁盘的访问时间的 10^{-6} ，快十万倍；从价格上，主存的每位价格约为磁盘的每位价格的 10^3 ，贵一千倍。所以，如果存贮层次能以接近于辅存的每位价格去构成大容量（等于辅存容量）的快速主存，这当然是提高性能价格比所最希望的。这样的体系不断地发展和完善，就逐步形成了现在广泛使用的虚拟存贮系统。对它，应用程序员可用机器指令的地址码对整个程序统一编址，如同具有对应这个地址码宽度，等于主—辅存贮层次存贮容量的虚（主）存空间，它当然可以比主存实际容量大得多，以致可以存得下整个程序。我们把这种指令地址码称为虚地址（虚存地址、虚拟地址）、逻辑地址、程序地址等，其对应的存贮容量称为虚存容量或程序空间，而把实际主存的地址称为物理地址、实（存）地址，其对应的存贮容量称为主存容量、实存容量或实（主）存空间。

目前大、中型机器的指令地址码可达 24~32 位；这样，虚存空间可达 16 兆字节至 8192 兆字节 ($2^{24} \sim 2^{32}$)，比实（主）存容量大得多，完全可以装得下绝大多数程序。

当用虚地址访问主存时，机器自动地把它经辅助硬件变换成主存实地址，并察看这个虚地址所对应的单元内容是否已经装入主存，如果在主存内就进行访问，如果不在主存内就经辅助软、硬件把它所在的那块程序由辅存调入主存，而后再进行访问。不论是虚、实地址的变换还是程序由辅存调入主存都不必由应用程序设计者安排，也就是说，这些本是实际存在的操作和辅助软、硬件，对应用程序设计者来讲是透明的。所有的存贮层次都必须满足这一点。

为了提高存贮层次的性能价格比，辅助软、硬件这部分成本只能占总成本中一个很小的比例。

这种虚拟存贮器的想法是 1961 年英国曼彻斯特大学的 Kilburn 等人提出来的，经过六十年代初到七十年代初的发展，这种虚拟存贮系统已被肯定下来。目前不光大、中型系统多数采用了虚拟存贮器，就是新设计的小型机也大都开始采用虚拟存贮器。

虽然，虚拟存贮器肯定是一个存贮体系，但存贮体系却不一定非得是虚拟存贮系统。例如，我国的 200 系列有存贮层次，第二章 § 2.1 讲过，它是用加界的方法把逻辑地址变换到物理地址，但它并不是虚拟存贮系统。虚拟存贮系统和非虚拟存贮系统本质的差别在于前者允许用户用比主存容量大得多的地址空间来访问主存，而后者用户只能以主存容量或以主存中被分配到的那部分空间（比主存容量小）来访问主存。还有，虚拟存贮系统每次访存时，都必须进行虚、实地址变换；而非虚拟存贮系统却不然。

作为主—辅存构成的存贮层次是起因于主存容量满足不了要求而提出来的，那么随着大规模集成电路存贮器在成本上的降低以及密度和速度上的提高，是否又会抛弃存贮体系的概念转到用很大容量的主存呢？前面讲过，主存、辅存并存的情况要长期存在，因此，存贮层次技术不仅不会衰亡，而且还会发展。当然，目前广泛使用的虚拟存贮系统必会进一步发展和完善。

上面是从容量需要引出存贮层次，下面则从速度要求来分析。前面已经讲过（参看图 5.2(a)），主存和 CPU 的速度差距已达 1 个数量级，为了解决这个速度差距对机器性能的影响，提出了多种办法。

一种办法是大家熟悉的，即在 CPU 中设置通用寄存器，很多运算可直接在 CPU 的通用寄存器中进行，从而减少与主存的交往，减轻速度差距的影响。但主存与通用寄存器间的传送要全由程序设计者安排，而且通用寄存器的数目不能过多，否则，它们的优化使用会过份增加编译程序的负担，降低效率。

另一种方法是采用存贮器的多体（模块）交叉并行存取来提高主存的等效速度，这在下面 § 1.3 并行存贮器还要详细讲。这种方法对速度的改善是有限的，较深的交叉存取所增加的设备量并不能使其性能相应成比例地提高，反而会使性能价格比显著下降。

以上这二种方法并不相互排斥，它们可以同时都被采用。

至于层次的方法则是高速缓冲存贮器方式，这是通用寄存器思想进一步发展的必然结果。它是在 CPU 和主存中间设置高速缓冲存贮器，构成高速缓冲—主存层次。要求 Cache 在速度上能跟得上运算器和主控制器的要求，在容量上要能存得下在一段时期内主控制器和运算器所要用的指令和数据。它经历了从便笺式到 Cache 的发展，其目标是怎样才能使高速缓冲存贮器内的指令和数据恰好是当时 CPU 所需要用到的。这种 Cache—主存层次如

图 5.8 所示。同“主—辅”层次一样，只是有了高速缓存并不能就说是有了高速缓存—主存层次，还需要有主存地址和缓存地址间的自动变换、映象和调度。Cache—主存间的地址映

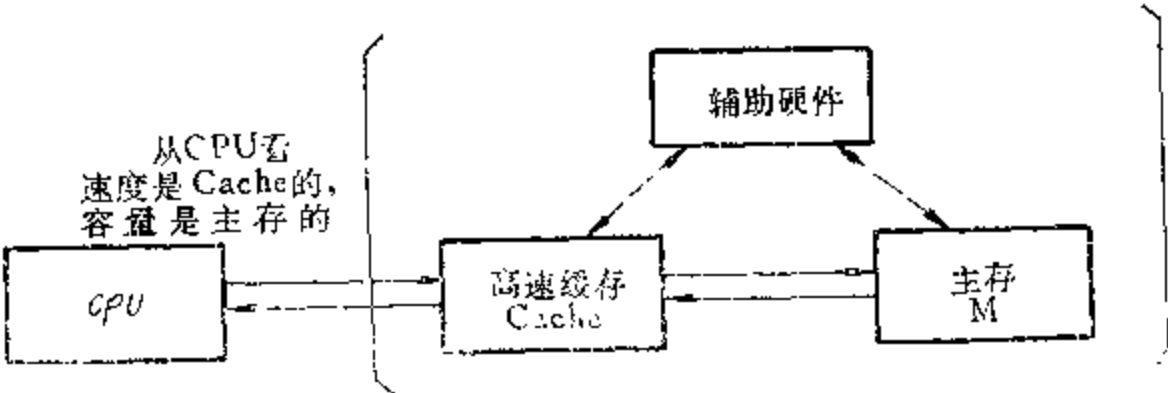


图 5.8 Cache—主存层次

象、变换和调度吸取了比它更早出现的主—辅虚拟存贮器的技术，在原理上是一样的，不同的只是速度要求高，因而不是软、硬结合而是全用硬件来实现。因此，Cache—主存层次不只是对应用程序员，就是对系统程序员也是透明的。

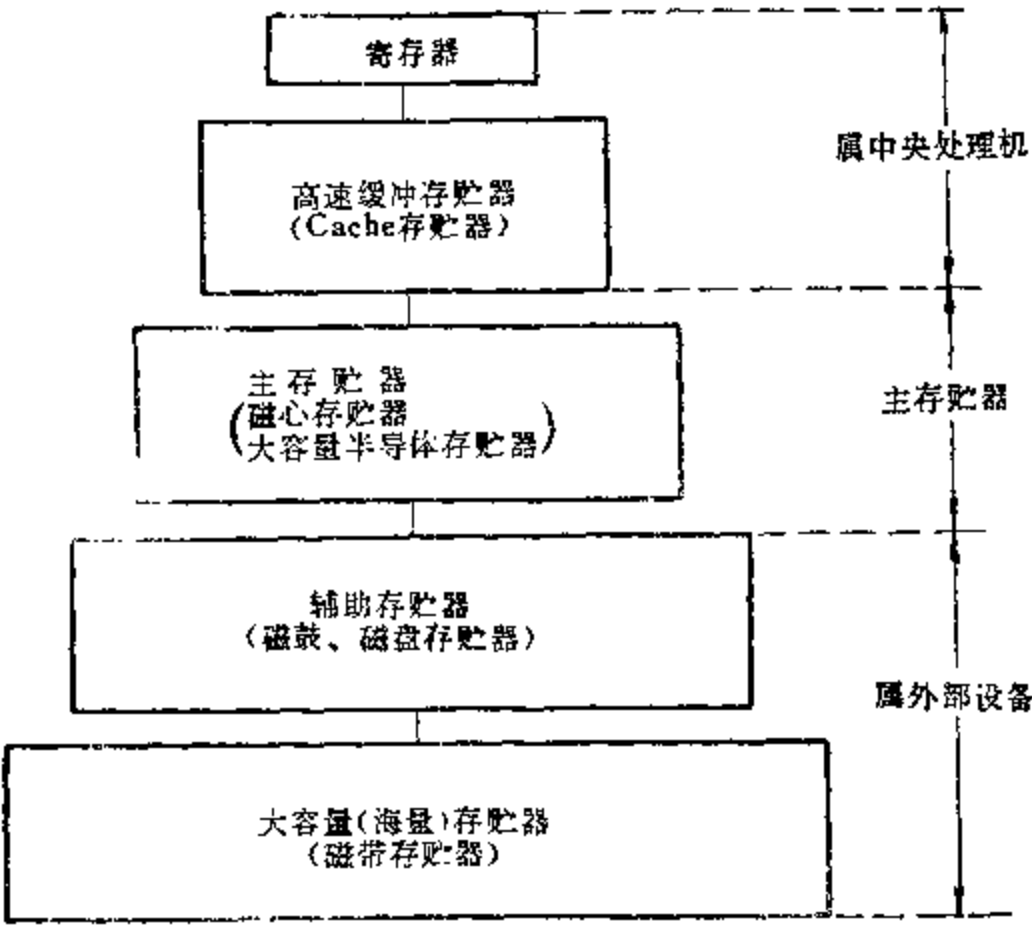
同理，Cache—主存层次的设计目标是要求从 CPU 来看，层次的速度是接近于 Cache 的，但容量是主存的，而每位平均价格应当接近于主存的。

为了使 Cache 存贮器与 CPU 在速度上能够相配，一般 Cache 存贮器采用与 CPU 同类型的半导体工艺。目前有很多的中大型机器都使用了 Cache 存贮器，小型机也在开始采用。

由于 Cache 与主存在速度、容量、价格上的差距不如主—辅存之间的大，在速度上，Cache 的存贮周期约为主存存贮周期的十分之一；在价格上，Cache 的每位价格是主存每位价格的十倍，因此，“Cache—主存”层次与“主存—辅存”层次虽然原理相同，但控制方式却有所不同。例如，主—辅层次只有辅存 \longleftrightarrow 主存 \longleftrightarrow CPU 的通路，没有 CPU \longleftrightarrow 辅存的直接通路；而对 Cache—主存层次除了有主存 \longleftrightarrow Cache \longleftrightarrow CPU 的通路外还有主存 \longleftrightarrow CPU 的直接通路。这些，我们以后还要详述。

以上讲了主存—辅存和 Cache—主存这两种二级存贮层次。主存—辅存存贮层次是为了弥补主存容量的不足，而 Cache—主存存贮层次是为了弥补主存速度的不足。虽然习惯上，只是主存—辅存存贮层次被称为“虚拟存贮”，但 Cache—主存存贮层次实际上也是基于相同的虚拟存贮原理。就是说，地址的映象、变换和替换原理都是一样的，可以统一分析。

由上述二级存贮层次当然可以推想到更多级的存贮层次（如图 5.9 所示）。它应该能把配置



于机器各个部分的各种存贮器有机地联系在一起。把属中央处理机的寄存器（有的机器不把它作为一层）、Cache 存贮器和主存贮器以及属外部设备的磁盘、磁带存贮器作为统一的层次整体来对待，力求用尽可能低的价格提供在速度上和容量上都满足要求的存贮系统。就是说，要求这种存贮层次作为一个整体，从运算控制部件看，能具有最高层的速度、最低层的容量。即对存贮系统的速度要求由最高层的昂贵器件满足，而对容量的要求由每位价格最便宜、但速度最慢的大容量存贮器提供。当然，还得要求程序和数据能按解题需要在各层之间传送，并且当运算、控制部件需要用到那一部分时，它应是已在最高层中准备好。

存贮层次的结构和操作系统的存贮管理密切相关，在这里，软、硬件的分界面是明显地相互交错。系统结构的设计者就是要联系具体的实现方法对这里的软、硬件功能分配进行仔细的分析确定。对存贮层次的研究一直是系统结构设计者的一个重要任务。进入七十年代后，在上述二级层次的基础上，不少机器，如 IBM370/168，Amdahl470/V6，IBM370/3030 等，都成功地把 Cache、主存和辅存构成了三级的存贮层次。

1.2-2 存贮体系的基本要求和性能评价

图 5.9 的多级存贮层次是由不同存贮技术、不同性能和不同价格的 M_1 、 M_2 、……、 M_n 级存贮器构成的层次。在这个层次中的每一级 M_i ，在某种意义上附属于较高一级 M_{i+1} ，CPU 和其它处理机直接与该层次的第一级 M_1 交往， M_1 再与 M_2 交往，如此等等，如图 5.10 所示。

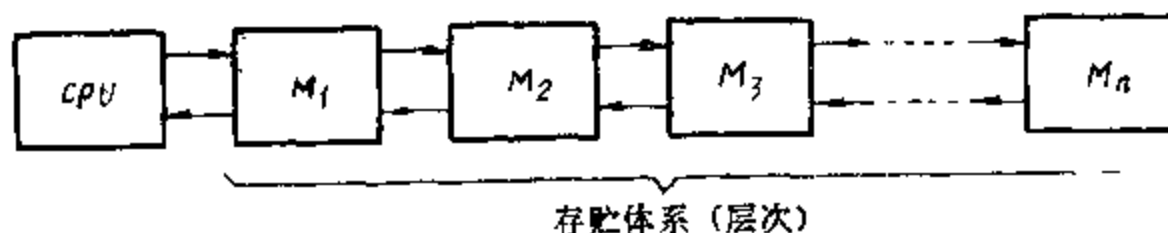


图 5.10 多级存贮层次的一般表示法

设 c_i 、 T_{A_i} 、 S_i 分别表示 M_i 的每位价格、访问时间和存贮容量，则 M_i 和 M_{i+1} 之间有：

每位价格	$c_i > c_{i+1}$
访问时间	$T_{A_i} < T_{A_{i+1}}$
容 量	$S_i < S_{i+1}$

因此，这个存贮层次的设计目标从 CPU 来看，使它在平均速度上接近 M_1 的，在存贮容量上是 M_n 的，而每位平均价格接近 M_n 的。但实际上，容量的目标是容易达到，速度的目标是较难达到，而价格的目标则更难达到。

在程序执行期间，CPU 产生一个连续的逻辑地址流，在任何时间这些地址总是以某种方式分布于存贮层次之内，它总能够被变换到某个 M_i 的物理实地址，如果 $i \neq 1$ ，则地址必须经逐级变换再赋值到 CPU 能直接访问的 M_1 。逻辑地址的这种定位通常需要在 M_i 和 M_1 之间进行相对较慢的信息传送。为了提高存贮层次的效率，当 CPU 要用到某个地址的内容时，总是希望它尽可能已在 M_1 中准备好，这就要求未来被访问信息的地址在某种程度上可以予知（判）。予判的准确性如何，往往是存贮层次设计好坏的主要标志，而这很大程度取决于

所使用的算法和地址映象、变换的方式。一旦出现被访问信息不在 M_1 中这种不希望的情况时，原先申请访存的程序暂停执行或被挂起，直到所需信息被调到 M_1 为止。然而这是讲的一般情况，若 M_1 为 Cache，则往往有某些特殊性，例如前述的 CPU 不只可与 Cache(M_1)，还可与主存 (M_2) 直接联系等。

为了要搞清存贮层次，需要搞清各种存贮器件的速度、容量和价格（前面已讲过），程序的定位（见 § 2），地址映象（见 § 3），地址变换（见 § 3，§ 5）和替换算法（见 § 4）等。存贮体系得以构成的一个前提是程序的局部性。另外，存贮保护和共享等（见 § 7）也是和存贮体系密切相关的。

用什么来评价存贮体系呢？为讨论简单起见，以二级体系 (M_1, M_2) 为基础来分析，主要引入每位价格、命中率 H 和访问时间 T_A 三个参数。参看图 5.11。

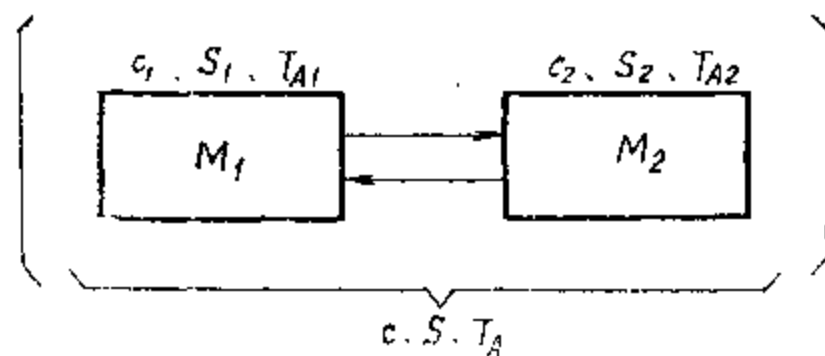


图 5.11 二级体系的评价

价格 c

存贮层次的每位平均价格为

$$c = \frac{c_1 S_1 + c_2 S_2}{S_1 + S_2} \quad (5.1-1)$$

其中 c_i 表示 M_i 的每位价格， S_i 表示 M_i 的以位计算的存贮容量。

为了达到使 c 接近于 c_2 的目标，应使 $S_1 \ll S_2$ 。但是实际上，如果相邻二级的容量差别很大，不只是实现级间地址映象和变换有较大困难，而且会降低层次的效率。

命中率 H

命中率是衡量存贮体系性能的重要参数。对这个二级存贮体系来说， H 定义为由 CPU 产生的逻辑地址能在 M_1 访问到（命中到）的概率。命中率可用实验方法求得，让一组有代表性的程序被执行或模拟，若逻辑地址流中能在 M_1 访问到的次数为 N_1 ，不能在 M_1 访问到，而当时在 M_2 还没调到 M_1 的次数为 N_2 ，则命中率

$$H = \frac{N_1}{N_1 + N_2} = \frac{N_1}{N} \quad (5.1-2)$$

命中率与地址判别算法（即判定下步会用到程序中的哪部份）以及 M_1 的容量有很大关系，我们当然希望 H 愈大愈好。

不命中率，或称失误率、脱靶率为：

$$\text{不命中率} = 1 - H \quad (5.1-3)$$

它是指的由 CPU 产生的逻辑地址在 M_1 中访问不到的概率。在一个分页系统（见 § 2）中，若 CPU 产生的逻辑地址所在页面不在 M_1 中就称发生“页面失效”，就是说，不命中就要发生页面失效。

访问时间 T_A

设 T_{A1} 和 T_{A2} 分别是 M_1 和 M_2 的访问时间，对 CPU 来说，访问层次的平均时间 T_A 为

$$T_A = H T_{A1} + (1 - H) T_{A2} \quad (5.1-4)$$

在大多数二级层次中，若所要访问的字不在 M_1 中，就必须由 M_2 把包括这个所需字的

一个信息块传送到 M_1 ，在这之后，CPU 才可在 M_1 中访问这个字。为传送信息块所需的时间 T_B 被称为块交换或块传送时间，因此有

$$T_{A_2} = T_B + T_{A_1} \quad (5.1-5)$$

代入 (5.1-4) 式，得

$$T_A = T_{A_1} + (1-H)T_B \quad (5.1-6)$$

对主存—辅存层次，信息块的传送需要一个相对慢得多的 I/O 操作，因此， T_B 比 T_{A_1} 大得多，即 $T_{A_2} \gg T_{A_1}$ 和 $T_{A_2} \approx T_B$ 。

设 $r = T_{A_2} / T_{A_1}$ 为存贮层次相邻二级的访问时间比， $e = T_{A_1} / T_A$ 为存贮层次的访问效率，我们总希望 T_A 愈接近 T_{A_1} 愈好，就是说，访问效率 e 越接近 1 越好。

由 (5.1-4)，我们可得

$$\begin{aligned} e = \frac{T_{A_1}}{T_A} &= \frac{T_{A_1}}{HT_{A_1} + (1-H)T_{A_2}} \\ &= \frac{1}{H + (1-H)r} = \frac{1}{r + (1-r)H} \end{aligned} \quad (5.1-7)$$

由 (5.1-7) 可获得 $e = f(r, H)$ 的关系如图 5.12 所示。其中 r 是取决于存贮器层次各级的器件和设备特性， H 与 M_1 容量及替换算法等许多因素有关。由图 5.12 可见：要使访问效率 e 愈接近于 1，在 r 值愈大时，就要求命中率 H 愈高。例如， $r = 100$ 时，为使 $e > 0.9$ ，必须使 $H > 0.998$ ；而当 $r = 2$ 时，则只需 $H > 0.889$ 即可。就是说，若 M_1 ， M_2 的速度差很大，要能具有足够高的访问效率，非得要有很高的 H 值不可。这样，对主存—辅存层次，由于其速度差达 10^4 ，那就要求命中率 H 很高，访问效率才能高。但 H 的提高是很不容易的，它受多种因素的影响。虽然减少主、辅存的容量差会提高 H 值，但这又与降低每位平均价格的要求相矛盾。若在主、辅存之间能增加一级（例如电子磁盘），使 r 值不要过大，则为获得同样的 e ，就可降低对 H 的要求。

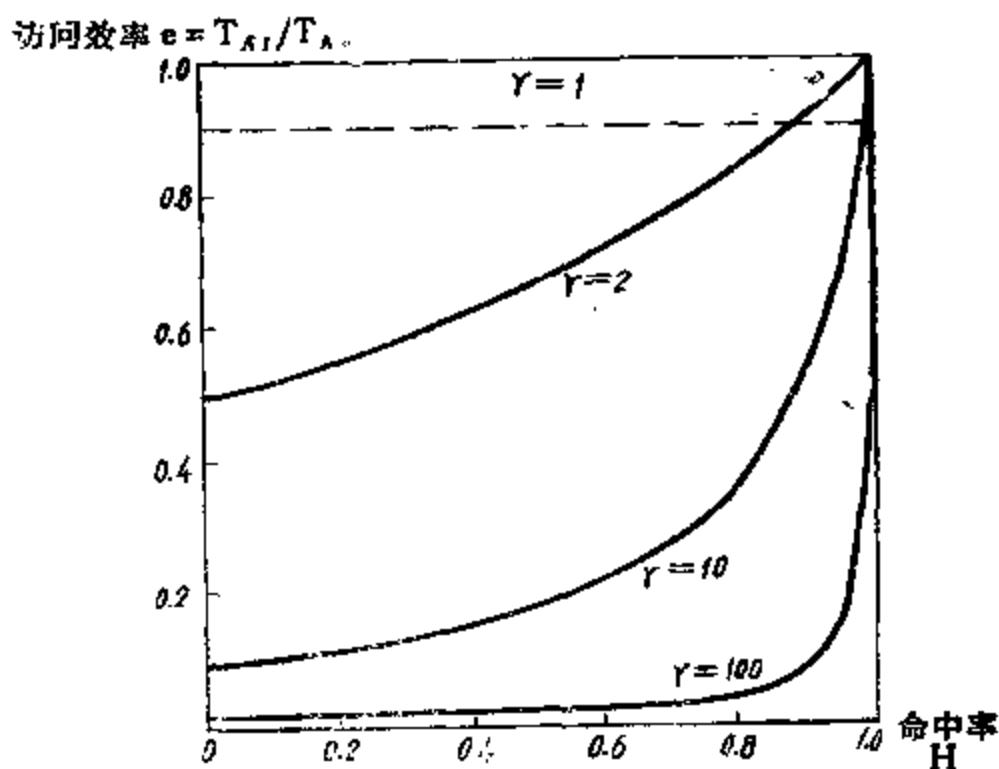


图 5.12 对于不同的 $r = \frac{T_{A_2}}{T_{A_1}}$ ，命中率 H 与访问效率 e 的关系

最后，简单讲讲存贮体系的透明性与其性能的关系。前面提过，主存和 Cache 之间的地址变换和成块调入调出并不需要程序员安排和干预，而完全由机器硬件自动实现，这对使用是方便的。

例如，PDP-11/45 是没有 Cache 的，PDP-11/70 是有 Cache 的，在 PDP-11/45 上运行的程序可以不必因为 PDP-11/70 有 Cache 存贮器而要加以修改后才能运行，只是采用了

Cache 后使速度有很大的提高。

主—Cache 层次不只是对应用程序员而且对系统程序员都是透明的，而主—辅层次，前已说过，对应用程序员是透明的，而对系统程序员来讲，这个层次基本上是不透明的，只是有些部分（如地址变换部分）是透明的，是由硬件直接实现的。

一般来说，系统结构的透明部份越多，程序员对它的使用就愈方便。然而，对存贮体系来讲，它愈透明，则程序设计者就会愈干予不了。控制不了这有时会不利于根据不同题目调节存贮体系的某些参数，因此，对存贮层次的透明性不能追求过分，而应是基本透明，但某些关键参数应能受程序员控制，以利于存贮体系的优化使用。

还有，存贮体系的透明性往往会给程序（任务）的切换带来麻烦，这是操作系统设计者和系统结构设计者所必须认识到的，这点我们在以后还要讲述。

1.3 并行主存系统

前已提过，采用存贮器多体（模块）交叉、并行存取能够提高主存的等效速度。对于给定的主存器件，是能够从结构上采取并行措施以提高主存系统速度的。这一节就来分析各种并行结构及其效率。严格地讲，这不属于存贮体系的内容，但是主存系统的结构和性能对存贮体系的构成很有影响，所以在进一步讲述存贮体系之前，有必要安排这一节。

1.3-1 并行访问多字

我们已经讲过，主存频宽 $B_m = \frac{w}{T_M}$ （位/秒），式中， T_M 为主存周期， w 为主存总线宽度（位数）， B_m 表示单位时间能通过存贮器总线送到处理机的信息位数。

为提高 B_m ，在同样的器件条件下（即同样的 T_M ），只有设法提高 w ，即让主存在一个存贮周期内不是读出一个字，而是读出顺序的多个字，如图 5.13(b) 所示是同时读出四个字。

因为能同时并行读出多个字，所以称为并行主存系统。如果读出的是指令，就可在一个主存周期取出多条顺序的指令（如图为四条），然后以一定顺序，例如以每隔四分之一主存周期（或处理机拍宽）从左到右依次送入指令（单字长）寄存器。这样，指令寄存器可以每

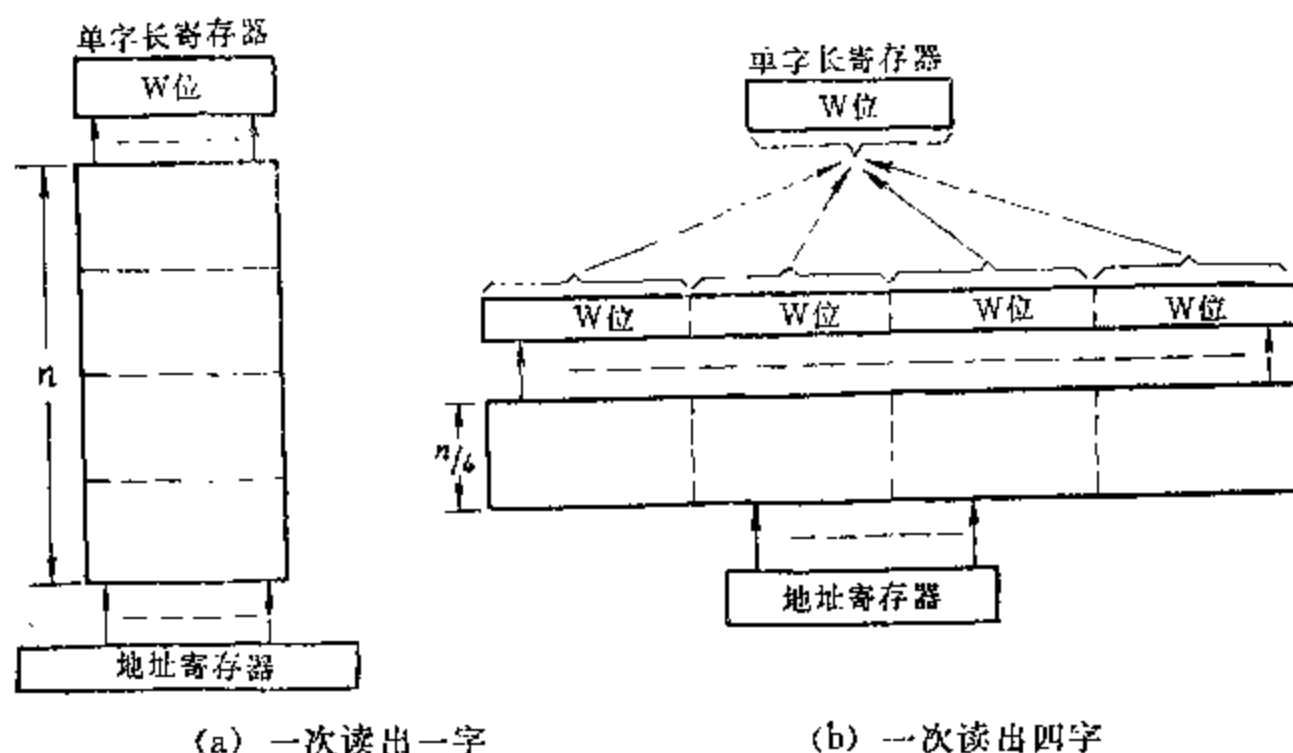


图 5.13 存贮器由一次读单字到读多字

隔四分之一主存周期取得顺序的一条指令，相当于取指令时间减少到四分之一，但主存的价格几乎仍然没变。这里，地址寄存器、译码电路均只一套，存贮体是单体，只是每个存贮单元包含了四个指令字。

对这种方式，如果所需的四个字正好是顺序的话，那么主存频宽确是提高到四倍，但若访问不是顺序的四个字，例如，遇到转移指令以及数据地址是随机分布时，实际的频宽就会显著下降。

1.3-2 m 体并行交叉存取

上述的并行访问多字方式是单体，一套地址寄存器和译码电路，读出的多字只能是顺序的多字。六十年代大型机的磁心主存就已采用这种结构；对它来说，由读单字到读多字会使读放及字驱动负载都按相应倍数增加。至于半导体存贮器，无非是用给定容量的存贮片子构成。虽然存贮片子上已经有了译码、读放等电路，但还得加上由逻辑片子组成的地址寄存器、译码器才能构成。然而，用存贮片子组合成一个大的存贮体和把它分成若干同等容量并具有各自的地址寄存器和译码电路的分体（例如为四个，见图5.14）这二种办法的价格差别很小。

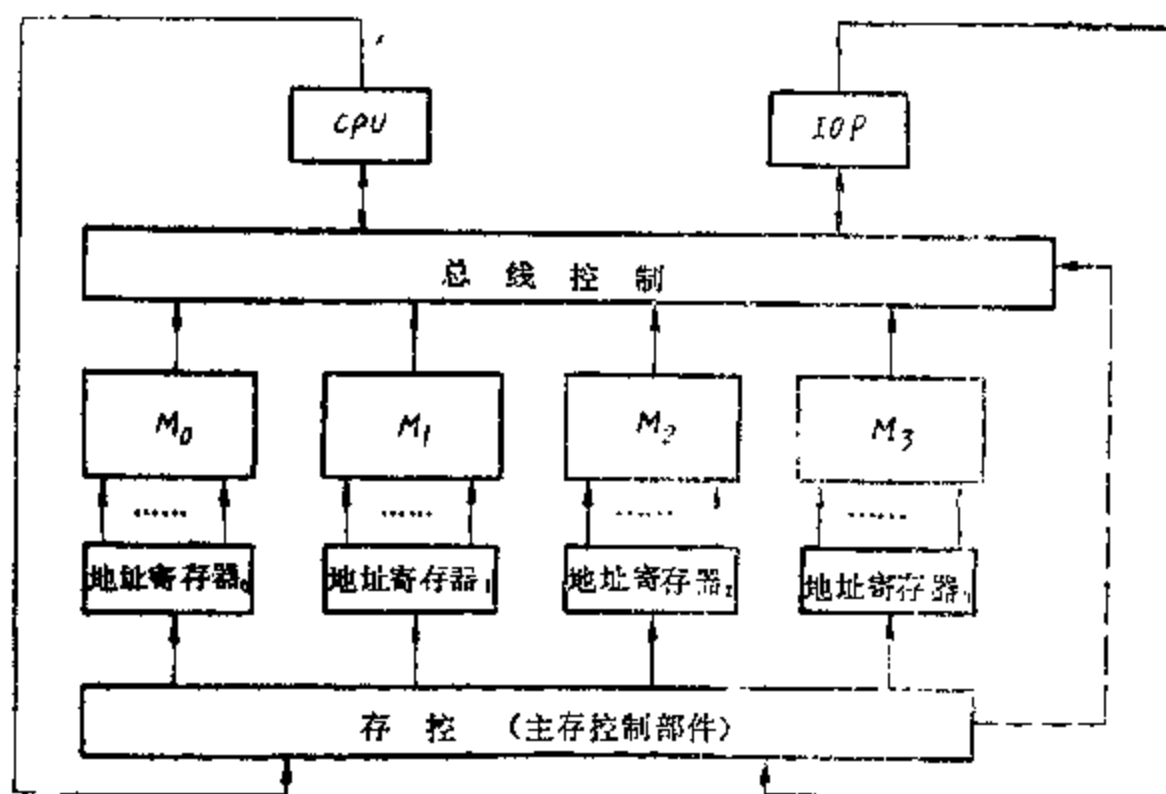


图 5.14 m 个分体并行存取 (m=4)

分体方式能实现多个分体并行存取；如果分体的宽度是单字长，则每个分体在一个存贮周期只能读出一个字，但多个分体的并行存取却能在一个存取周期内并行读出多字。虽然分体方式的最大频宽与单体多字方式的相同，但是，它比单体多字灵活。因为并行读出的四个字不象单体多字那样非得是顺序的，而是只要这四个字分属于不同的分体即可。这个灵活性很有好处，它使这种方式实际的频宽会比单体多字方式的要高，这点下面还要讲到。当然这四个分体的输出应经“总线控制”再送到 CPU 或 IOP，由存贮器控制器（存控）来控制 和安排 CPU、IOP 来的访存申请的先后排队。目前，所谓并行主存系统多是指的这种包含多个能独立编址的分体存贮系统。当然对磁心存贮器也可采用这种分体方式，但所化设备要比单体多字的。

分体方式的时间关系可以有二种处理办法。一种是让四个分体同时启动（读或写），使

得四个分体在一个存贮周期内同时被访问，但经存控控制分时使用总线。另一种是四个分体分时启动（每经四分之一存贮周期可启动一个分体），如图 5.15 所示（当然运行中每个分体相邻二次启动的实际间隔可能会大于一个存贮周期，它取决于地址流的状况）。这二种办法只是具体工程实现上的差别，它们都是多体并行存取。目前一般采用分时读出法，它对缩短总线的宽度和便于控制及以存控部件的工程实现都是有利的。它相当于最高每隔 $\frac{1}{m}$ （此例为 $1/4$ ）个主存周期，主存系统可流出或流入一个字。就是说，对主存系统来讲，仿佛是有串地址流流入，而有一串信息流流出的流水结构，很适于和流水式中央处理机的联系。

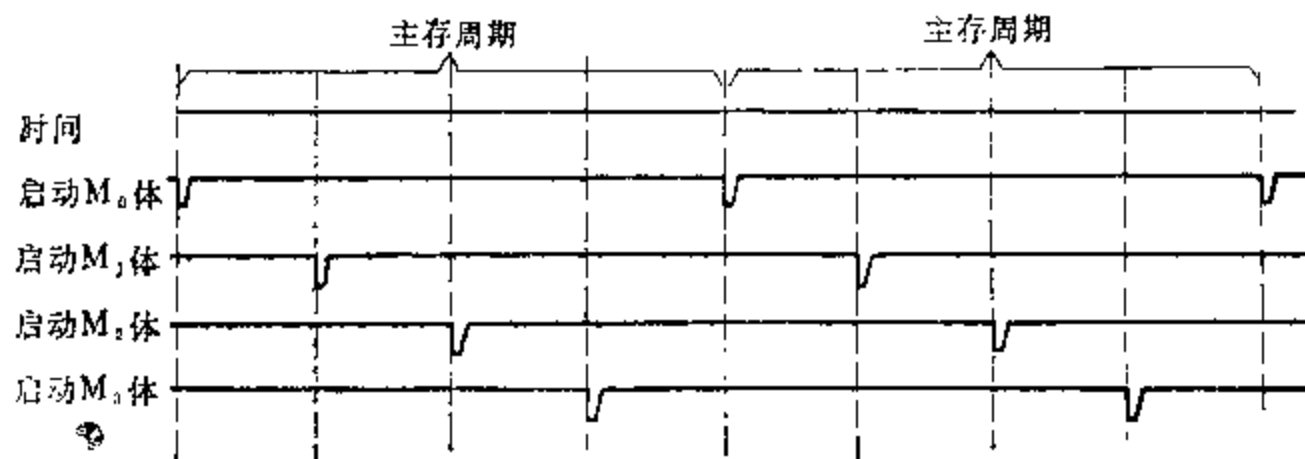


图 5.15 四个分体分时启动的时间关系

显然，并行存贮体分体越多（即 m 越大），相当于主存频宽愈宽，主存系统的信息流量就愈可能与处理机所需的信息流量相匹配。

对于多 CPU 系统，并行主存系统还可以经纵横多路开关（如图 5.16 所示）以及树形结构等与各 CPU 和 IOP 发生联系。这些在第七章还要详述。

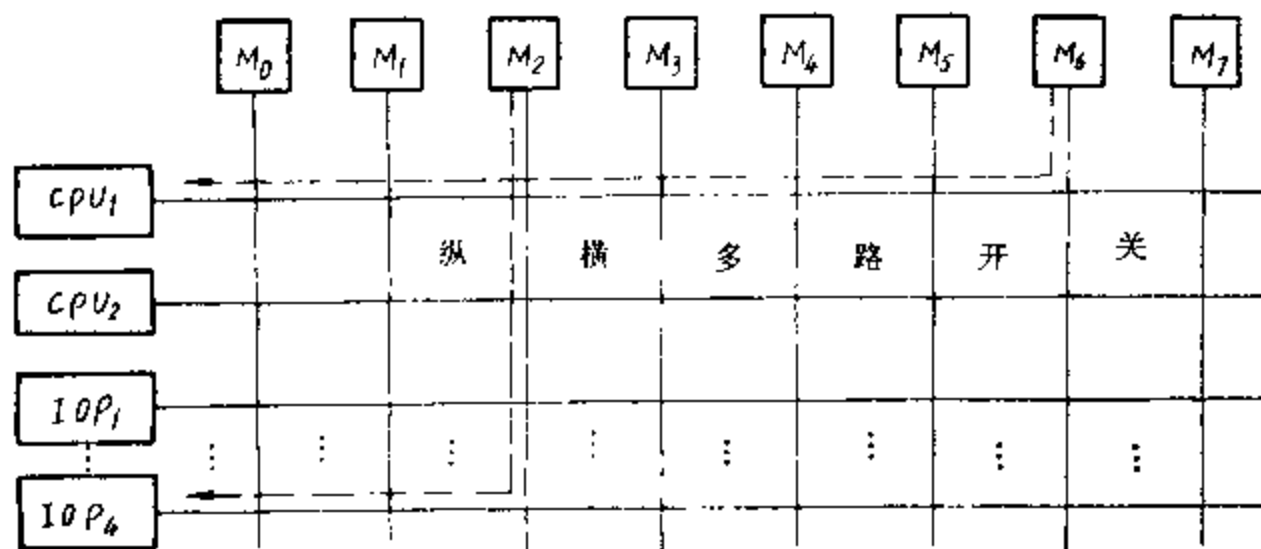


图 5.16 并行主存系统经纵横开关转接

这里，我们主要考虑单处理机的情况。各分体间的编址采用多体交叉方式。例如，若一个主存系统由四个分体 M_0, M_1, M_2, M_3 组成，则各个分体的编址序列如表 5.4 所示。

单体容量为 n 的 m 个分体交叉编址，称为模 m 交叉（表 5.4 为模四交叉）。 M_j 体的地址编址模式为

$$mi + j,$$

其中

$$i = 0, 1, 2, \dots, n-1; j = 0, 1, 2, \dots, m-1.$$

存贮器总的地址码长度为 $\log_2 n + \log_2 m = \log_2 n \cdot m$ ，总的存贮容量为 $n \cdot m$ 。注意，前述图 5.13(b)的单体四字方式也是采用模 m 交叉编址，只不过它只有一个地址寄存器，因此是

表 5.4 地址的模四交叉编址

模 体	地 址 编 址 序 列	对应二进制地址码最末二位的状态
M_0	0, 4, 8, 12, ..., $4i + 0$, ...	0 0
M_1	1, 5, 9, 13, ..., $4i + 1$, ...	0 1
M_2	2, 6, 10, 14, ..., $4i + 2$, ...	1 0
M_3	3, 7, 11, 15, ..., $4i + 3$, ...	1 1

利用它的末二位，如表 5.4 所示，对四字宽的数据寄存器选出对应给定地址的字。

模 m 交叉编址是单体多字和多分体并行存取这二种方式都采用的。对于顺序访问，频宽都可提高到 m 倍。例如，对 $m=4$ 时，若要访问的地址是 0、1、2、3 或 8、9、10、11 等，则二种方式的频宽都可提高到四倍；然而，若地址是 1、4、10、15（参看表 5.4），则对单体多字就不能同时读出这四个字，而需要用四个主存周期才能全部读出，因而其实际频宽毫无提高，如同单体单字的一样。但对多分体并行存取，由于这四个地址分属于四个分体，因而照样可以并行存取，使频宽仍可提高到四倍。只是若四个地址中，有二个以上同属一个分体（如 1、5、10、14），则频宽的提高实际达不到 m 倍（ $m=4$ ），这时称为发生分体冲突（因为 1、5 地址同属 M_1 分体，10、14 地址同属 M_2 分体，发生了冲突，不能同时读出），要化二个主存周期才能将这四个字读出（访问 1、10 地址需一个主存周期，而访问 5、14 地址又需另一个主存周期）。显然，由于多分体并行存取比单体多字的灵活性大，因此它的实际频宽更有可能接近于最大频宽。

如果将多分体并行存取与单体多字结合在一起，即每个分体的宽度不是单字，而是多字，就能进一步提高频宽。

目前不仅是大型机，就是小型机也愈来愈多地采用并行主存系统。例如，小型机中的 PDP-11，可以用两个分体并行（可以并行读出分属二个分体但不一定是顺序的二个单元）。大中型机中的 IBM370，一个字有四个字节，其中 370/135 不用并行技术，主存周期约 800~990 毫微秒，每次读出 2 个字节，其频宽略大于 2 字节/微秒；而 370/165 采用四个分体并行，加上每个分体一次读二个字，虽然存贮器存贮周期为 2 微秒，但其频宽最高却可达：

分体数 \times 分体宽度（字数） \times 每字字节数/主存周期 = $4 \times 2 \times 4/2 = 16$ 字节/微秒，相比之下差到八倍。有的机器采用更大的 m 值，例如 CDC-6600，采用模 32 交叉，主存周期为 1 微秒，最大频宽本可达 32 字/微秒，但实际只按 10 字/微秒的频宽使用，CDC-7600 也是模 32 交叉，主存周期为 0.275 微秒，同样只能按每个周期能访问 10 个字使用，实际等效存贮周期为 27.5 毫微秒。

1.3-3 模 m 的大小与转移概率 λ 及吞吐量的关系

由 CDC-6600、7600 的例子可以看出，采用模 32 交叉，对于存贮器存贮周期为 1 微秒来说，频宽本应该是 32 字/微秒，然而实际上却只达 10 字/微秒，仅为理想改进的三分之一都不到。就是说，并行主存系统的实际效率并不象所希望的那么高，这是为什么呢？

首先，是工程实现方面的问题。所有模块中数据的写入和读出都要通过数据总线，并行

交叉程度越高，总线上并联的负载就愈重，为此需要相应采取很多的措施，而且引线会增长，使传输延迟增加。

其次，是系统效率的问题。对模 m 交叉，取顺序的指令，效率可提高到 m 倍。但实际上程序中指令不总是顺序执行的，一旦出现转移，效率就会下降，转移的频度越高，这种并行主存系统的效率下降就越大，加上考虑到数据的顺序性比指令差，因此就不能按理想的最高频宽来使用。

现在，通过一个模型来作些分析，以便从中取得有益的结论。

对有 m 个独立分体的主存系统，设处理机发出的是一串地址为 A_1, A_2, \dots, A_q 的访存申请队，在每一个主存周期之前，这个申请队被扫描，并且截取从队头起的 A_1, A_2, \dots, A_k 序列，称为申请序列，它定义为在 K 个地址中没有二个或二个以上的地址处在同一分体中的最长序列。就是说，这个申请序列 $A_1 \sim A_k$ 不一定是顺序编址的，只要求它们之间不会引起分体冲突。很明显， K 最大可以为 m ，然而由于可能发生分体冲突，所以它往往会小于 m 。截取的这个长度为 K 的申请序列只能同时访问 K 个分体，因此这个系统的效率取决于 K 的平均值， K 越接近 m ，效率就越高。

设 $P(K)$ 表示申请序列长度为 K 的概率密度函数，其中 $K = 1, 2, \dots, m$ 。即 $P(1)$ 是 $K = 1$ 的概率， $P(2)$ 是 $K = 2$ 的概率， $P(m)$ 是 $K = m$ 的概率。 K 的平均值用 B_m 表示，则

$$B_m = \sum_{k=1}^m K \cdot P(K) \quad (5.1-8)$$

它实际上就是每个主存周期所能访问的平均字数，即实际的主存频宽 B_m 。 $P(K)$ 与程序的状况密切相关，如果访存申请队都是指令的话，那么影响最大的是转移概率 λ ，它定义为给定指令的下条指令地址为非顺序地址的概率。指令在程序中一般是顺序执行的，但遇到成功转移，则申请序列中在转移指令之后与它在同一存贮周期读出的其它顺序单元内容是没有用的；而且，即使转向去址与转移指令不产生分体冲突，也由于处理机响应时间来不及，不可能与转移指令安排在同一个存贮周期内访存。因此，申请队中，如果第一条就碰到转移指令，且转移成功，那与第一条指令并行读出的其它 $m-1$ 条指令是没有用的，即相当于 $K = 1$ ，所以， $P(1) = \lambda$ ；对于第二条碰到转移指令并转移成功，即 $K = 2$ 的概率自然是第一条没有转移（其概率为 $1 - \lambda$ ），第二条是转移指令且转移成功的情况，所以， $P(2) = [1 - P(1)] \cdot \lambda = (1 - \lambda) \cdot \lambda$ ；同理， $P(3) = [1 - P(1) - P(2)] \cdot \lambda = (1 - \lambda)^2 \cdot \lambda$ ；如此类推， $P(K) = (1 - \lambda)^{K-1} \cdot \lambda$ ，其中 $1 < K < m$ 。如果前 $m-1$ 条均不转移，则第 m 条是否转移都是 K 等于 m ，故 $P(m) = (1 - \lambda)^{m-1}$ 。

将 $P(1), P(2), \dots, P(m)$ 代入 (5.1-8) 式，得

$$B_m = 1 \cdot \lambda + 2 \cdot (1 - \lambda)\lambda + 3 \cdot (1 - \lambda)^2\lambda + \dots + (m - 1)(1 - \lambda)^{m-2} \cdot \lambda + m(1 - \lambda)^{m-1} \quad (5.1-9)$$

用数学归纳法可得

$$B_m = \sum_{i=0}^{m-1} (1 - \lambda)^i$$

这是一个等比级数，因此

$$B_m = \frac{1 - (1 - \lambda)^m}{\lambda} \quad (5.1-10)$$

可见, 每条指令都是转移指令且转移成功 ($\lambda = 1$) 时, $B_m = 1$, 并行存取的好处全部消失, 如同使用单体单字存储器一样, 所有指令都不转移 ($\lambda = 0$) 时, $B_m = m$ (即 B_m 的最大值), 效率最高。

图5.17 画出 m 为 4、8、16 时 B_m 与 λ 的关系曲线。看出, 如果转移概率 λ 在某个范围之外, 例如 $\lambda > 0.3$ 时, $m = 4, 8, 16$ 的 B_m 差别不大, 即在这种情况下, 模 m 取值再大, 对系统效率也并没有带来多大的好处; 而在 $\lambda < 0.1$ 时, m 值的大小对 B_m 的改进则有显著的影响。换句话说, 因为程序的转移概率不会很低, 靠加大 m 值来提高并行主存系统的频宽是有限度的, 而且性能价格比会随 m 值的加大而下降。

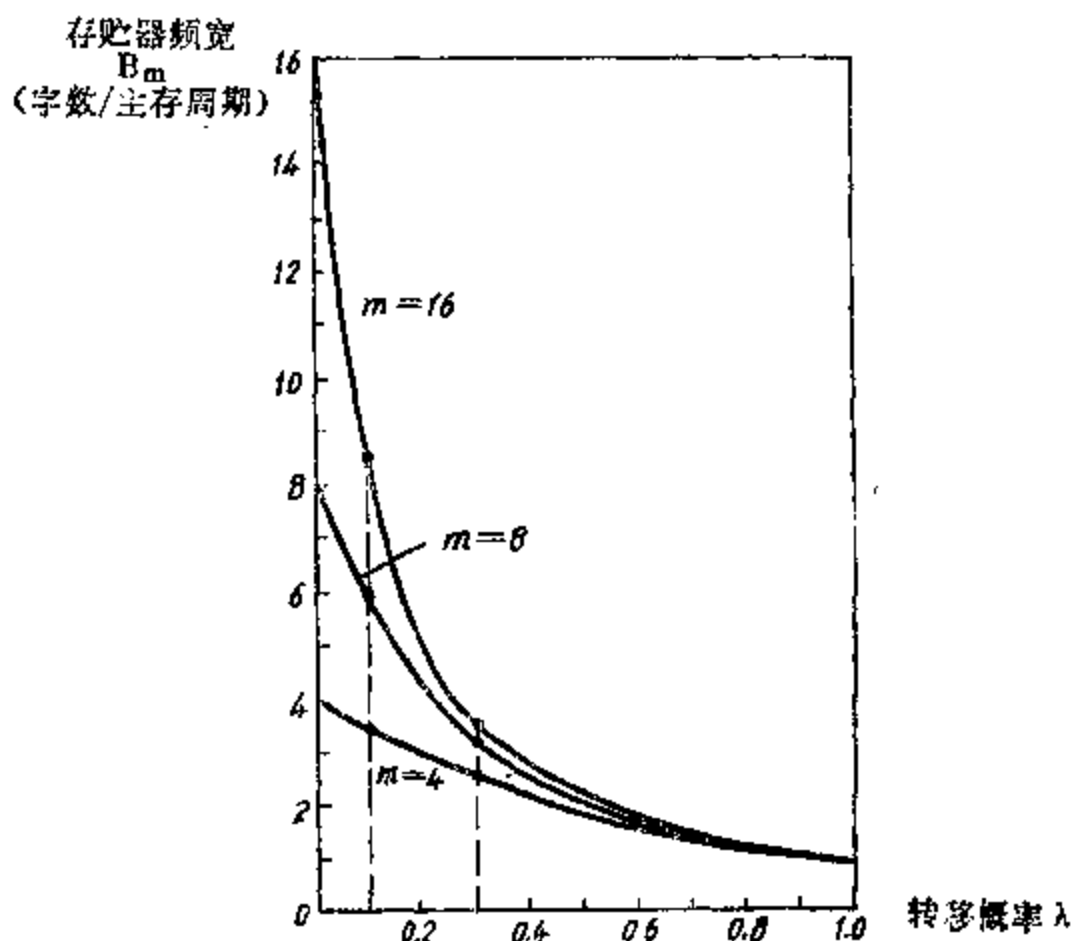


图 5.17 m 个分体并行存取的 $B_m = f(\lambda)$ 曲线

一般认为, 在通常情况下, 采用 $m = 16$ 意义已不是太大, 目前大多取 $m = 2 \sim 8$ 。如果主存频宽仍然满足不了需要, 那就不如加用 Cache—主存层次, 而不要单纯靠加大 m 值来解决。当然, 在有了 Cache—主存层次时, 仍然需要采用并行主存系统, 以便能快速地往 Cache 送信息, 这点, 我们以后还要讲。

至于数据, 由于它的分布随机性比指令的大, 单纯靠 m 值的加大, 是更难以满足处理机所需要的信息流量。

如果所有的申请 (包括指令和数据) 都是完全随机性的, 用单服务、先来先服务排队论模型进行模拟, 可求得 m 的取值与主存的频宽关系如下:

m	1	2	4	8	16
B_m (字/主存周期)	1.0	1.5	2.219	3.245	4.704

可以看出, 随着 m 的增加, B_m 是以比 m 小得多的 $m^{0.56}$ 来增大 (或更为准确地用 $\sqrt{\pi m/2} - 0.28$ 来表示)。即对随机访问, 主存频宽与 m 的关系是近似于所谓平方根关系。当然, 指令流肯定不会是完全随机性的, 就是数据流也不会全是随机性的, 例如阵列、表格等就会是顺序存取。因此, 总的来看 B_m 的值总是会比平方根值大。

采用第三章关于流水方式中的缓冲技术, 不是遵循先来先申请的原则, 而是以不引起分体冲突为出发点, 则并行主存系统的实际流量还会增加。

通过这一节的分析, 使我们认识到应从系统结构的观点来分析机器各个部件的性能。应尽可能构成分析模型, 从中引出关系式或关系曲线, 以指导设计。决不能盲目设计, 片面追求某些部件的高指标, 而是要寻求整体的相对平衡。当然对计算机的分析比较复杂, 要定量

分析尤其难，至今还没有很适合的数学工具，不过排队论还是可用的数学工具之一。

1.3-4 多端存储器

以往的存储器从概念上讲只有一个读端与写端，要么读，要么写，读、写不能同时进行，因此也就只需要一套地址寄存器和译码电路，如图 5.18(a) 所示。随着半导体技术的发展，我们可以给一个存储体配上多套地址寄存器、译码电路构成所谓多端存储器。它实质上也是并行存储器中的一种。

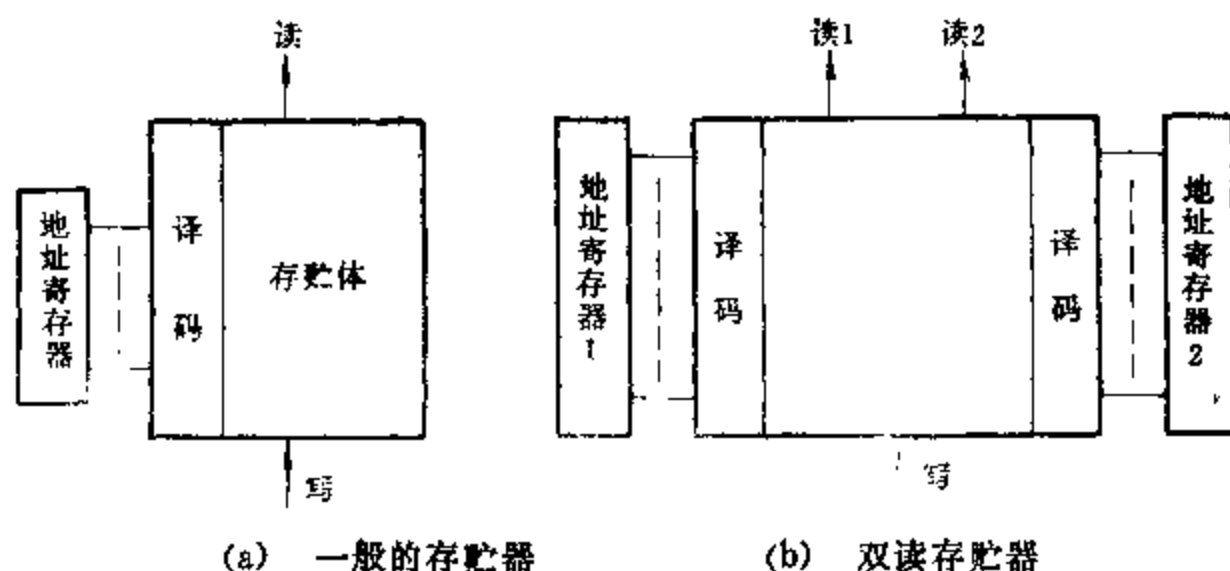


图 5.18 多端（双读）存储器

例如，可以构成“双读”存储器，如图 5.18(b)，它具有二套地址寄存器和二套译码电路，可以同时读出该存储器中任意二个地址单元中的内容。它没有前述单体多字法中必须是顺序的字以及多分体并行存取法中，必须是分属不同的分体的字那些限制。

当然也可以构成“双写”存储器，或“读写同时”进行，等等。显然，这种多端并行存储器结构会给存储体系的构成带来灵活性和效率的提高。这种多端存储器已应用在某些微处理机的通用寄存器中，而且在并行处理中也已被采用。

1.4 相联存储器

此小节本不属于系统结构的内容，但考虑到相联存储器在存储体系的地址变换以及在优先权排队等应用中很有用，故在此作一简介。

一般的随机访问存储器都是按地址访问，这在硬件的实现上是直接和方便的，但对某些应用，却会带来不方便。

存储单元（64 位）内容					
地址编号					
1	张	三	224	22	陕 西
2	李	四	126	20	北 京
3	王	五	240	23	上 海
4	陈	六	178	19	北 京
5	姚	七	026	24	北 京
...

姓名字段 学生证号字段 年龄字段 籍贯字段

图 5.19 存放在存储器中的学生登记表

举例来说,存放如图 5.19 那样的学生登记表,每个存贮字(例如 64 位二进制)有四个字段,分别表示学生的姓名、学生证号、年龄和籍贯。

在回答象“陈六的学生证号是多少?”这样的问题时,如果采用按地址访问存贮器,就必须指明对应陈六的存贮单元的物理地址 4,通过读出第 4 号地址单元而查到他的学生证号。但是,地址 4 与名字字段“陈六”并没有逻辑上的联系,因此需要通过费时的程序方法来解决。一种直观的方法是按地址顺序把逐个单元的内容读出,然后将它们的姓名字段与存放在某个寄存器的内容(陈六)相比较,直到相符。显然,这种顺序检索需许多个存贮周期,而且它随陈六所在地址不同而有很大差异。那能否从寄存器的内容“陈六”直接访问到 4 号单元呢?即能否由原来按地址访问改为按内容来访问呢?

相联存贮器就是具有这种功能的硬件,因此又称为按内容访问存贮器。它的构成原理见图 5.20。它要求:

(1) 有一个按它存放的内容(如陈六)进行检索的输入寄存器。在硬件上寄存器的输出能同时与存贮体所有字按位比较。有一个掩蔽寄存器把不参与检索的字段掩蔽掉,存贮体内对应被掩蔽字段位置的各位不参加比较。

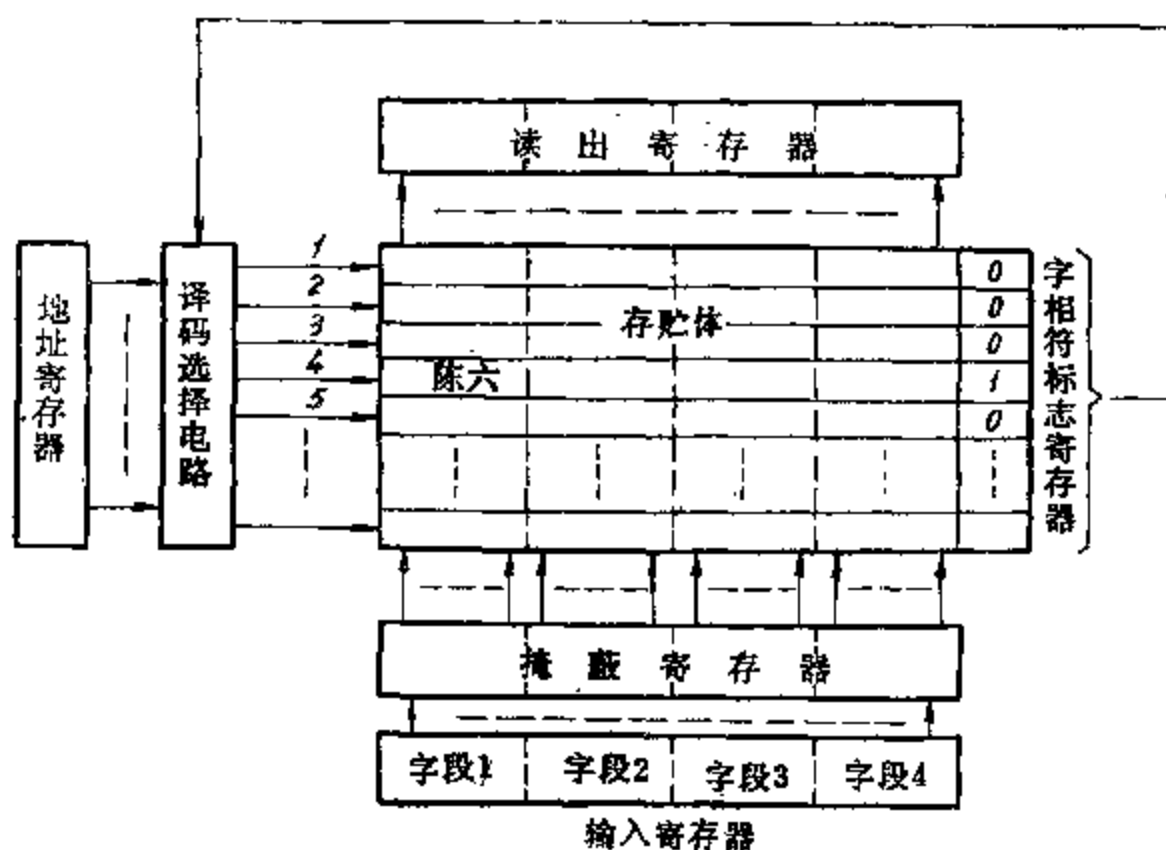


图 5.20 相联存贮器的构成原理图

(2) 每个存贮单元的每位都有“位相符”标志,凡是与输入寄存器的相应位比较相符的位,其“位相符”标志位为“1”,而当某个存贮单元内的所有位相符标志都是“1”(对应被掩蔽字段的各位,其位相符位也为“1”)时,就给出“字相符”标志,存入字相符标志寄存器。

(3) 由字相符标志控制译码选择电路,把相应的字读出(图中,字相符标志寄存器的第 4 位为“1”,由它控制把第 4 号单元读出)。

相联存贮器一般既有按内容访问的能力也有按地址访问的能力,所以相联存贮器仍然需有地址寄存器和译码选择电路。

可用下式进行位相符比较,

$$F_{mn} = S_{mn}D_{en} + \overline{S_{mn}}\overline{D_{en}}$$

其中 D_e 表示输入位, S 表示被比较的位, m 表示字号, n 表示位号。若比较是位相符, 则 F_{mn} 为“1”。

这样就由“读4号地址单元的内容”改为“读姓名为陈六的存贮单元的内容”。可见按内容访问的功能能实现所有字的同时并行比较。

每个存贮单元的内容可看成是由键符和数据二部分组成:

键	符	数	据
---	---	---	---

键符是要与输入寄存器未掩蔽部分相比较的字段。上例中, 键符为学生的姓名, 其它部分为数据。一个字中哪个部分是键符由掩蔽寄存器指明。这样, 若检索要求改为“籍贯是上海的同学叫什么名字?” 那只需把键符改到籍贯字段即可。

参看图 5.19, 如果问: “籍贯是北京的同学叫什么名字?” 那么答案就应是李四、陈六、姚七三个人, 即字相符标志出现 3 个 1, 这就要按某种预定好的顺序 (一般是按地址顺序) 依次读出这些单元的内容。

此外, 也有并不要求读出内容的情况, 例如, “籍贯是北京的同学有几名?” 只需由掩蔽寄存器指明参与比较的是“籍贯”字段, 输入寄存器存放的键符为北京, 启动存贮器后对字相符标志寄存器中“1”的个数进行计数即可求得。

相联存贮器还可进行大于、小于、是否处于给定上下界范围之内、求最大值、最小值、次大值、次小值及其它各种类型的逻辑检索, 为此只需在每个单元增加相应的比较逻辑。相联存贮器的每个单元, 不仅能存贮, 还能进行逻辑判断, 所以也称为“分布逻辑存贮器”。当然也因此比只有存贮功能的按地址访问存贮器复杂和昂贵。显然, 通过比较找出所需字的时间要比按地址访问一个字的时间要长。

早在五十年代中期就试图用低温超导元件构成相联存贮器, 但只有到七十年代有了大规模集成电路之后才在经济上成为可用。七十年代, 由于价格仍然较高, 而且容量大后, 速度过分降低, 因此只局限于应用在必须快速检索, 但所需容量较小的地方。例如, 用于存贮层次中的地址映象和变换等。另外, 它在并行处理中也很有用, 例如由它构成相联处理机等。

按内容访问, 相联存贮器是用全硬实现, 至于软的方法有散列法、表结构法等, 这些方法也可以软硬结合实现, 散列法在 § 3 将会讲到。

§ 2 程序的局部性与定位

2.1 程序的局部性和工作区

本节内容原不属系统结构范围, 但因和存贮体系的构成关系密切, 所以在此作一简介。

前面讲过, 为使存贮体系能有效地工作, 总希望处理机要访存的内容尽可能已在速度最快的 M_1 中; 因此能否予判(知)出下步所要访问的程序块对存贮体系的构成是非常重要的, 而这种予判的可能性是基于计算机程序的一个特性, 即程序的局部性。在一段时间之内典型的程序所要用地地址是趋向于集中在较小的范围之内, 在给定主存字被处理之后, 下一个需要处理的指令或数据极大可能是紧挨着该主存字或是在该主存字附近的区域之内。就是说, 程序执行时的地址分布不会是随机的, 而是簇聚成自然的块或页面 (存贮器中较小的连续单元区)。

程序之具有局部性首先是因为程序本身总是顺序编写和编址，并总是线性地展开在物理存储器中，实际执行时基本也是如此顺序的，只是遇到转移指令时才会跳到一般是远离该区域的另一区域。然而跳去之处通常是子程序，因此在一段时间内又会簇聚在该区域进行。而后往往会再跳回到开始的区域之中。如果把访问主存的物理字地址作为时间的函数，如下的地址分布例子是有典型性的：

时 间	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
所访问的存储器字的地址(十进制)	1	2	3	5	8	10	2072	2073	2074	2079	2080	12	17	13	32
	主程序(第0块或页)						转子程序(第2块或页)				跳回主程序(第0块或页)				

其次是因为存在程序的循环，在循环中的语句要被重复地执行，使得访问到某些地址的频率会较高。

对数据来说也是如此。计算机的数据不是随机分散地存放，一般总是以向量、阵列、树形、表格等形式存放，其中每类数据都是簇聚地存贮。

以上是从时间分布来分析程序的局部性，如果从地址空间的访问频度来看，会出现图 5.21 的情况，看出，地址空间的访问频度极不均匀。程序局部性的这种空间簇聚使我们认

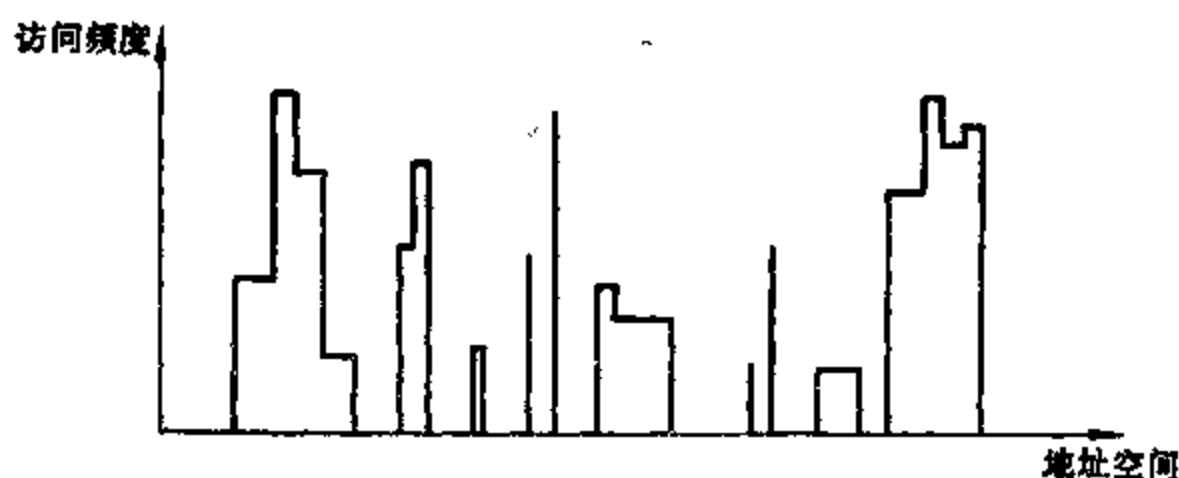


图 5.21 访问地址的非随机分布

识到，如果做得到把访问频度高的那些块或页始终存放在存贮层次最高一级，那必然会获得高的访问效率。还应看到，被称为一块或一页的容量比起整个程序或主存容量一般要小得多。也就是说，我们并不需要使层次的 M_1 级能存得下整个程序，而只需存得下访问频度高的那些块或页就行。然而，问题在于机器怎么能够知道，哪些块、页是用得最频繁的呢？

一种办法是利用程序局部性的下述特点，即对任意时间 T ，可以通过 $T - \Delta T$ 到 T 的时间内执行的程序地址来予判 T 到 $T + \Delta T$ 时间内程序的指令和数据将用到的地址的范围。我们称从 $T - \Delta T$ 到 T 这段时间所访问的地址范围为 $T - \Delta T$ 到 T 时间的工作区。

举例来说，如果一个程序空间等分成六个区（或称为页） $P_1 \sim P_6$ ，那么在时间轴上看，这六个区的典型使用情况可能如下：

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
所访问的页 P_i	P_2	P_1	P_6	P_1	P_3	P_2	P_6	P_4	P_5	P_4	P_5	P_2	P_4	P_2	P_5	

可见：

(1) 工作区随 ΔT 的取值而变。若把每个时间间隔定为 Δt ，这样，从 $T = 15$ 往回看，

其工作区随 ΔT 取几个 Δt 而不同:

ΔT	工 作 区
$1\Delta t$	P_5
$2\Delta t$	P_5, P_2
$3\Delta t \sim 8\Delta t$	P_5, P_2, P_4
$9\Delta t \sim 10\Delta t$	P_5, P_2, P_4, P_6
$11\Delta t$	P_5, P_2, P_4, P_6, P_3
$12\Delta t \sim 15\Delta t$	$P_5, P_2, P_4, P_6, P_3, P_1$

(2) 工作区在一段时间之内是很可能不变的。例如对 $T=8$, 取 $\Delta T=3\Delta t$, 其工作区为 $P_5P_4P_6P_2$, 那直到 $T=14$, 它一直适用。

(3) 工作区是要随 T 值的变化而变, 然而不是突然的, 即从 $T-\Delta T \sim T$ 到 $T \sim T+\Delta T$ 其工作区内容的变化只是部分的。例如, 对 $T=7$ 那点, 取 $\Delta T=6\Delta t$, 其工作区从 $P_4P_6P_2P_3P_1$, 变到 $P_5P_4P_2$, 只变化了 P_6 一页。而且 ΔT 取值越大, 变化的内容所占比例越小, 也就是说, 包含 N 页的工作区变化到完全不同的另外的 N 个页要经过比访问 N 个页面长得多的时间。

存贮体系的得以建立以及命中率的提高就是基于上述程序局部性的特点。

本例中, 从 $T=15$ 往回看, 对 $\Delta T=15\Delta t$, 各页按其访问过的先后次序排列为

$P_6, P_2, P_4, P_6, P_3, P_1$

我们把 P_1 称为在 ΔT 期间内最久未被访问过的页面, 把 P_6 称为在 ΔT 期间内最近被访问过的页面。这个概念在 §4 讲替换算法时, 还要详述。

2.2 程序的定位

在第二章 §2 已讲过, 中央处理机只能执行已装入主存的程序。把在辅存中的程序调入主存, 要进行程序的定位。为了方便地使用机器, 要求程序的定位对应程序员来讲是透明的, 就是说, 他不必顾及如何把逻辑地址变换成实际的主存物理地址。

程序定位可以全部或部分地在程序生成的各个不同阶段或程序执行时实现:

(1) 程序员在编写程序时进行: 这在早期没有操作系统和定位辅助硬件时是如此。它一般不允许不同用户分享同一块主存空间, 目前极少采用;

(2) 在程序编译时由编译程序来进行: 用户只指明程序中的名称地址, 由编译程序直接变换成实存物理地址;

(3) 在编译好(或汇编好)的程序装入主存时由装入程序进行;

(4) 在程序运行时由操作系统进行。

我们知道, 不论单道或多道程序, 在程序执行过程中, 一个时期内, 往往只需有一部分是在主存内。随着程序的执行, 程序的各块会在主存和辅存之间调进调出。此外, 为了提高主存空间的利用率, 要求能及时地释放主存中的无用区。这些都需要有能自动把程序的逻辑地址变换为物理实地址的定位辅助硬件机构以及确定程序在主存中位置的算法。

目前的定位机构主要有加基址(界)方式和地址映象方式两种。前一种方式已在第二章

§2 讲过。而地址映象方式，由于是基于下一节要讲的分段与分页，所以在 §3 中再细述。

对于加基址法，如图 5.22 所示，地址形成逻辑根据指令的规定（如变址等）形成逻辑

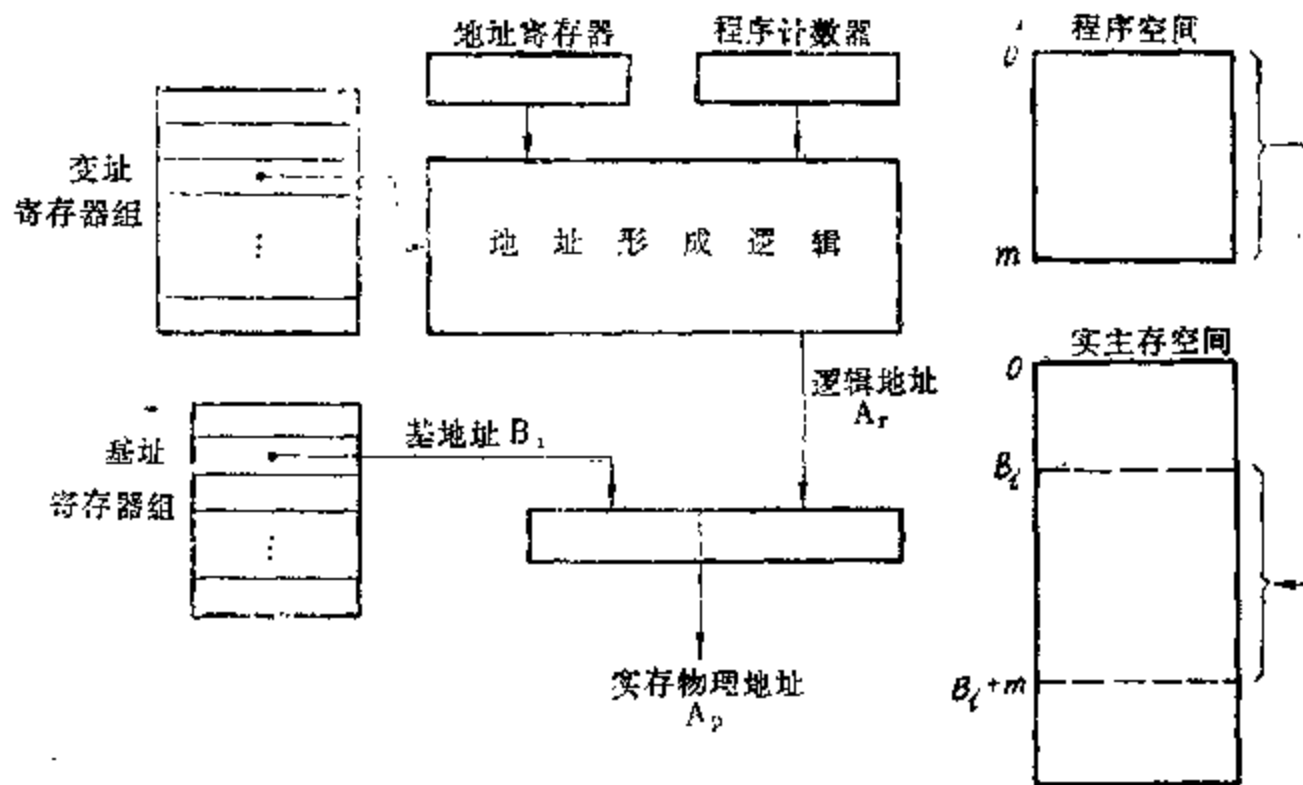


图 5.22 基址浮动法再定位

地址 A_r ，它再与 B_i 复合形成主存实地址 A_p ，这种复合可以采用拼接的办法，把 B_i 作为 A_p 的高位部分，而把 A_r 作为 A_p 的低位部分（如图 5.22 所示）；也可以采用第二章 §2 的 $B_i + A_r = A_p$ 的相加办法。拼的方法一般并不比相加方法省时间，因为加 B_i 的操作往往可以和 A_r 的形成合并进行，如果地址形成是经专用加法器实现，且与指令的执行重迭进行，则对机器速度不会有什么影响。

在第二章 §2 已经讲过静态再定位与动态再定位的概念。采用动态再定位，能提高主存的利用率。举例来说，

设主存内 A、B 两道程序的位置和大小如图 5.23 所示。若有 D 道程序，其长度 $n \leq m$ ，原先考虑是 A 道比 B 道先执行完，因此是让 D 道进入 A 道的程序位置，所以 D 道的基址值原定为 a 。但是，实际执行过程中若是 B 道先于 A 道结束，且 B 道程序等于或大于 D 道程序，则 D 道本可进入原存 B 道的空间与 A 道共同运行。然而，若采用的是静态再定位，那只有等 A 道执行完后才能进入原分配给 D 道的位置。这既降低了主存的利用率，也延缓了 D 道程序的执行。若采用动态再定位，则只需使 D 道程序的基址值由 a 变为 $a + m + 1$ 则就克服了上述缺点。动态再定位中基址值的修改应当用特权指令来执行。要注意到，加界方法不便于实现多个任务对子程序的共享，而下面要讲到的段、页地址映象方法就易于实现这点。

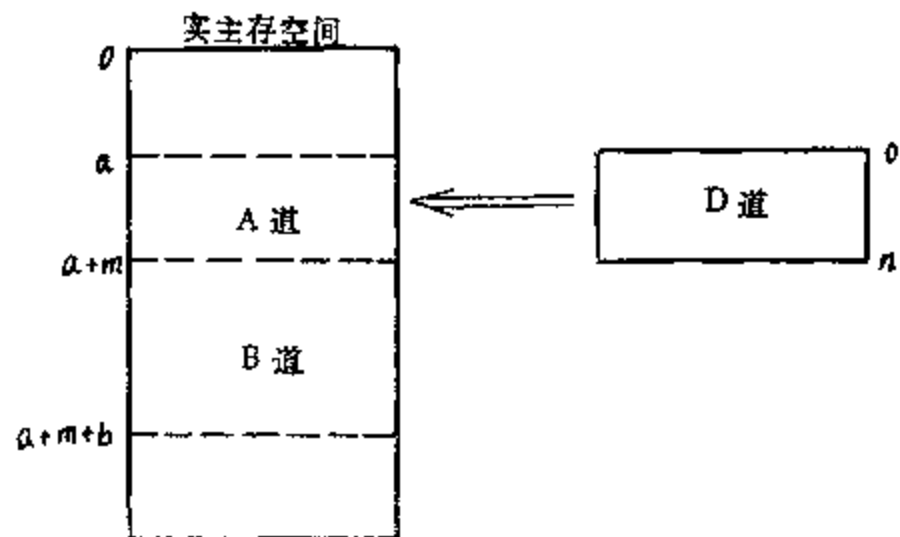


图 5.23 用基址浮动动态再定位

2.3 分段与分页

我们在这里着重于从提高主存利用率来讲述。

2.3-1 段式

早期，一个完整的程序是从头到尾连续编址，这种方法有很多问题。

首先，会使程序的编制时间过长。这是因为整个程序只能从头到尾串行编制，否则地址定不下来；然而，各个功能相对独立的部分本来是可以并行编制的。另外，对不定长的那些表、栈、队等只能按很大的保险量编址，这会使整个程序空间不合理地过分增大。

其次，会降低主存（这是很宝贵的资源）的利用率。这有二个含义。一是指的这种编址方法希望是整个程序装入主存（这种编址方法给用复盖方法分块装入带来很多麻烦），但任何时候活跃的只是一部分，其它部分白占据着主存位置，造成主存浪费。二是指的主存另头的浪费。所谓另头，如图 5.24 所示，是指主存中已用区域之间的空隙。由于程序是连续存贮，尽管主存中各个另头加起来比要调入的程序大，但是只要每个另头都比要调入的程序小，那这个程序还是调不进去，这既使另头白白浪费，又使这个程序延缓执行。虽然可以采用主存的重新分配，使另头集中、合并，也可能腾得出存得下要调入程序的空间，但这要将已用部分搬家，很费时间。

为解决上述问题就出现了按块、分段编址的方法。

本来一个程序是可以由多个逻辑上形成整体，相互独立或基本独立，且可清楚定义的模块所组成。这些模块可以是各种能赋予名称的子程序或分程序，例如 2-10 转换、三角函数、编译程序、装入程序等等；也可以是表格、数组、树、向量、可变长数据阵列等数据结构。

这些模块的大小是不同的。而且，有些模块的大小还可能是预先不知道的。例如，在程序的编译过程中会出现的各种编译用表格，如符号表、常数表、语法分析树、堆栈、队等，它们往往随编译过程不断扩大。因此需占用多少程序空间有时是很难事先确定的。因而，若使每个模块是一个段，每段都各自从 0 编址，采用分段编址，就会给程序的分段调入主存提供方便。在同一时期，只需把那些活跃段调入运行即可，这样，就可减少每个程序所需的主存空间。而且当要调入的一块程序，若其大小超过了已有各个另头，如图 5.25 所示，却小于主存

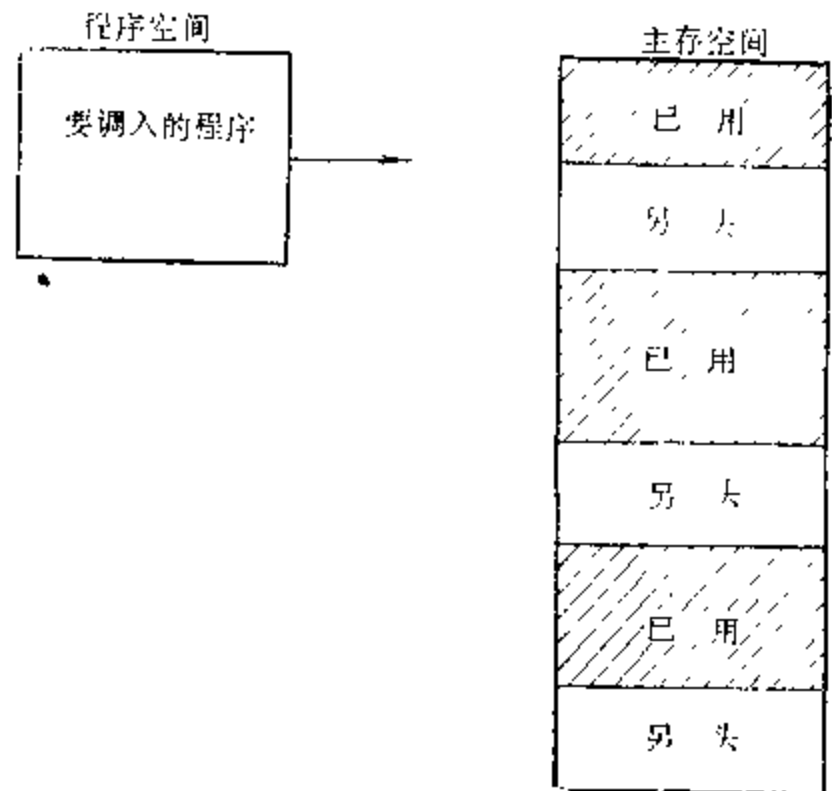


图 5.24 主存另头的浪费

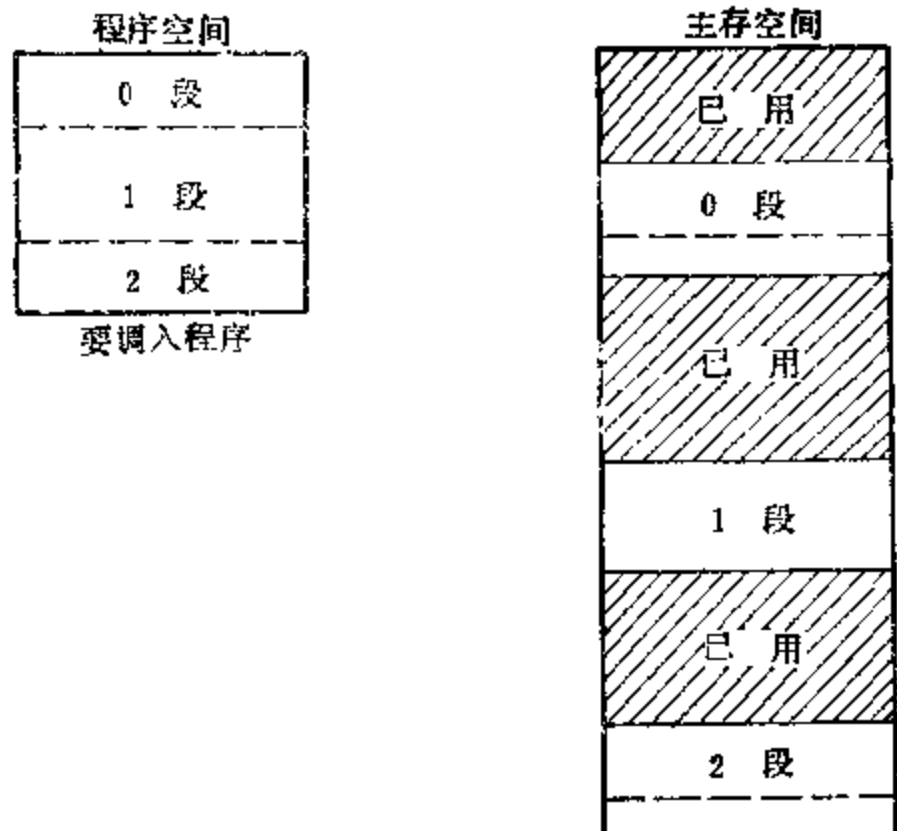


图 5.25 分段法会减少另头的浪费

中另头的总和时，只要这块程序的各段比这些另头小时，就可以通过分段装入主存另头（如图所示，0、1、2 段分别装入三个另头）的办法达到能运行更多道程序或用较小的主存容量来运行同样多的程序的目的。

由于每段是相对该段起点从 0 编址，在调入该段时，只需由操作系统赋予该段一个基址（即该段在主存中的起始地址），由基址和段内原编的地址相组合就能形成该段每个单元在主存中的实地址。

把主存按段分配的存贮管理方式称为段式管理。为此，通常需要有“段表”来指明各段在主存中的位置，如图 5.26 所示。每段都有它的名称（用户名称或数据结构格式名称或段号）。段起点是指明该段的起始位置（即基址值）。装入位表示该段是否已被装入主存，

“1”表示已装入，“0”表示未装入。装入位为“0”时的段起点没有意义。显然各段的装入位状态是随该段是否活跃而变化的。段长是指明该段的长度以实现访问地址是否越界的判断。当然，段表还可以有其它的内容。段表本身也是一个段，可以存在辅存，在需要时再调入主存，但一般是驻留在主存内。

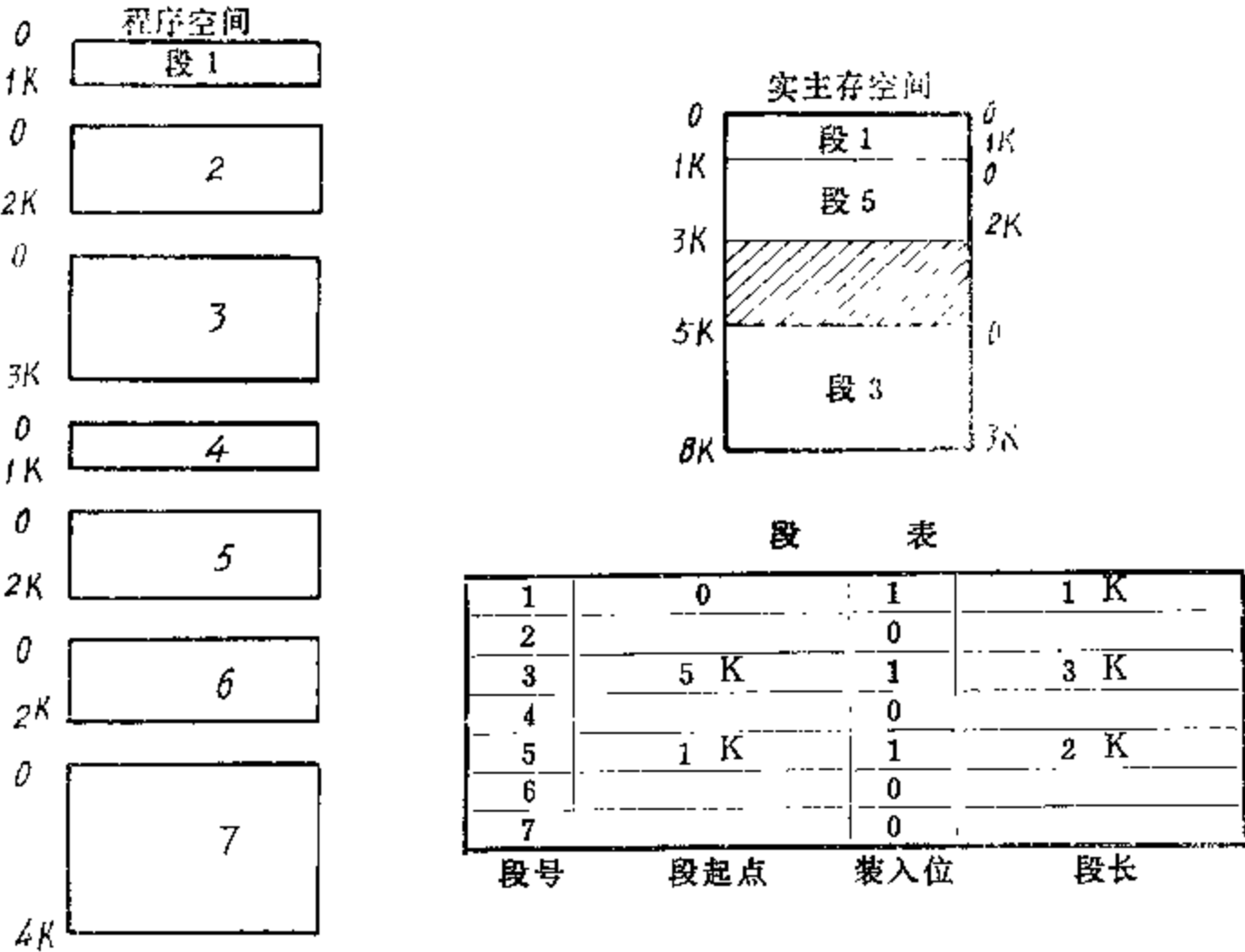


图 5.26 用段表进行段式管理

分段方法能使大程序分模块编制，从而可多个程序员并行工作，缩短编制时间。在执行或编译过程中不断变化的可变长段也便于处理。各个段的修改、增添，并不影响其它各段的编址，各用户以段的链接形成的程序空间可与主存实际容量无关。

分段还便于几道程序共用已在主存内的程序和数据，如编译程序、各种子程序、各种数据和装入程序等就不必在主存中重复存贮，只需把它们按段存贮，并在几道程序的段表中有公用段的名称及同样的基址值即可。

由于各段是按其逻辑特点组合的，容易以段为单位实现存贮保护。例如，可以安排成常

数段只能读不能写；浮点数段只能读或写，不能按指令执行；子程序段只能执行，不能修改；有的过程段不能写以防破坏，也不能读以防被“盗”，只能执行，如此等等。一旦违反规定，就中断，这对发现程序设计错误和非法使用是很有用的。

还要注意到，段式管理和第二章 § 1.3 讲的描述符技术在基本概念上是一致的。但描述符方法是从应用角度，从数据表示出发；而我们在这里讲段式管理是从主存管理，从提高主存利用率出发。

2.3-2 页式

段式管理对主存来讲，由于各段长度不同，会给主存管理带来麻烦，虽然主存的“另头”浪费有了减少(如图 5.25 所示)，但仍然有可能较大。举例来说，主存内已有 A、B、C 三个模块，其大小和位置如图 5.27 所示，若程序空间有一长度为 12K 的 D 模块要调入，采用段式管理时，虽然 D 模块小于主存所有另头的总和 (16K)，但因为没有一个另头装得下

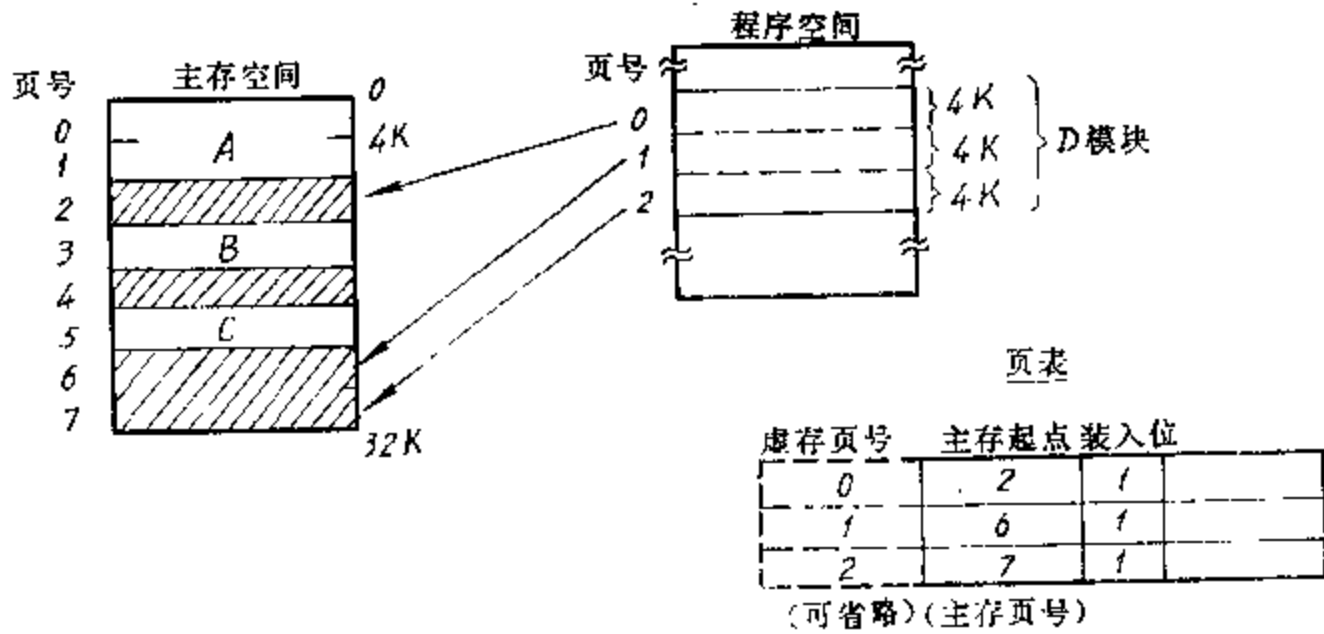


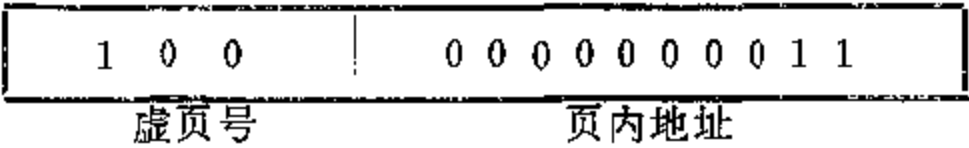
图 5.27 页式存贮

它，所以仍无法装入。为减少这种浪费，就提出了页式管理，它把主存空间和辅存或虚存中的程序空间都与程序的编制无关地机械等分成固定大小的页（页面大小随机器而异，一般在 512 到几千字节）。主存按页顺序编号，而每个独立编址的程序空间（可以是一个程序，也可以是一个模块）有自己的页号顺序。若每页为 4K 字节，则独立编址的 D 模块有三页长，且有自己的页号（0~2），而各个页可以装入主存中不同的页面位置，如图 5.27 所示。如同段式管理需有段表，页式管理也需有页表。类似地可以想到页表应有程序空间中页面的页号（也称虚存页号）；每页在主存中的起点，表示该页是否已经装入主存的装入位及其它内容。但是，段表中段名（号）一般是有其程序上的逻辑含义，而程序空间的虚页号却纯粹是顺序的编号，由于虚页号与页表的行号必是对应的（如虚页第 2 号必与第 3 行对应），因此虚页号项可以取消。又由于主存页号与对应此页号的起始位置是对应的，主存起点也可用主存页号来代替，如图 5.27 所示。页表实际上是主存页号与程序空间页号的映象表。显然，页表的建立不应由程序员进行，即对程序员来说应是完全透明的，它是由操作系统和存贮体系配合，根据主存运行情况来建立。主、虚存页面之间的映象方式有好多种，一种方式是每个虚页可以进入主存的任意页面，只要该页面没被占用。关于映象方式在 § 3 还要详述。

当程序各页在主存中定位之后，如何根据页表由逻辑地址变换到主存的实地址呢？我们知道，一个程序(模块)是统一编址，如图 5.28 的 A 程序是从 0 到 8K 编址。但每个地址 是可以用虚页号和页内地址来表示，例如程序地址：

100 0000000011

可以用虚页号和页内地址表示为



若程序的始点必须和主存页面的始点一致，则当第 4 号虚页是定位在主存的第 1 页时，这个单元在主存的地址应是

01 0000000011

因此，为实现虚、实地址变换，只需将虚页号通过页表变换成相应的实页号，而页内地址原封不动地照搬拼接即可形成该字的实存地址。变换过程由图 5.28 可以看清。

虽然我们是从减少主存另头浪费引出页式管理，然而它也并不能完全消除另头。因为程序的大小不可能恰好是页面大小的整数倍，如果某程序为 10K，而每页是 4K 大小，则这个程序调入主存，需占用三个页面，其中必有一页中还有 2K 的位置没有用上，我们把这种页内另头称之为内另头。因为程序的始点必须是主存页面的始点，不然就不能用上述那样简便

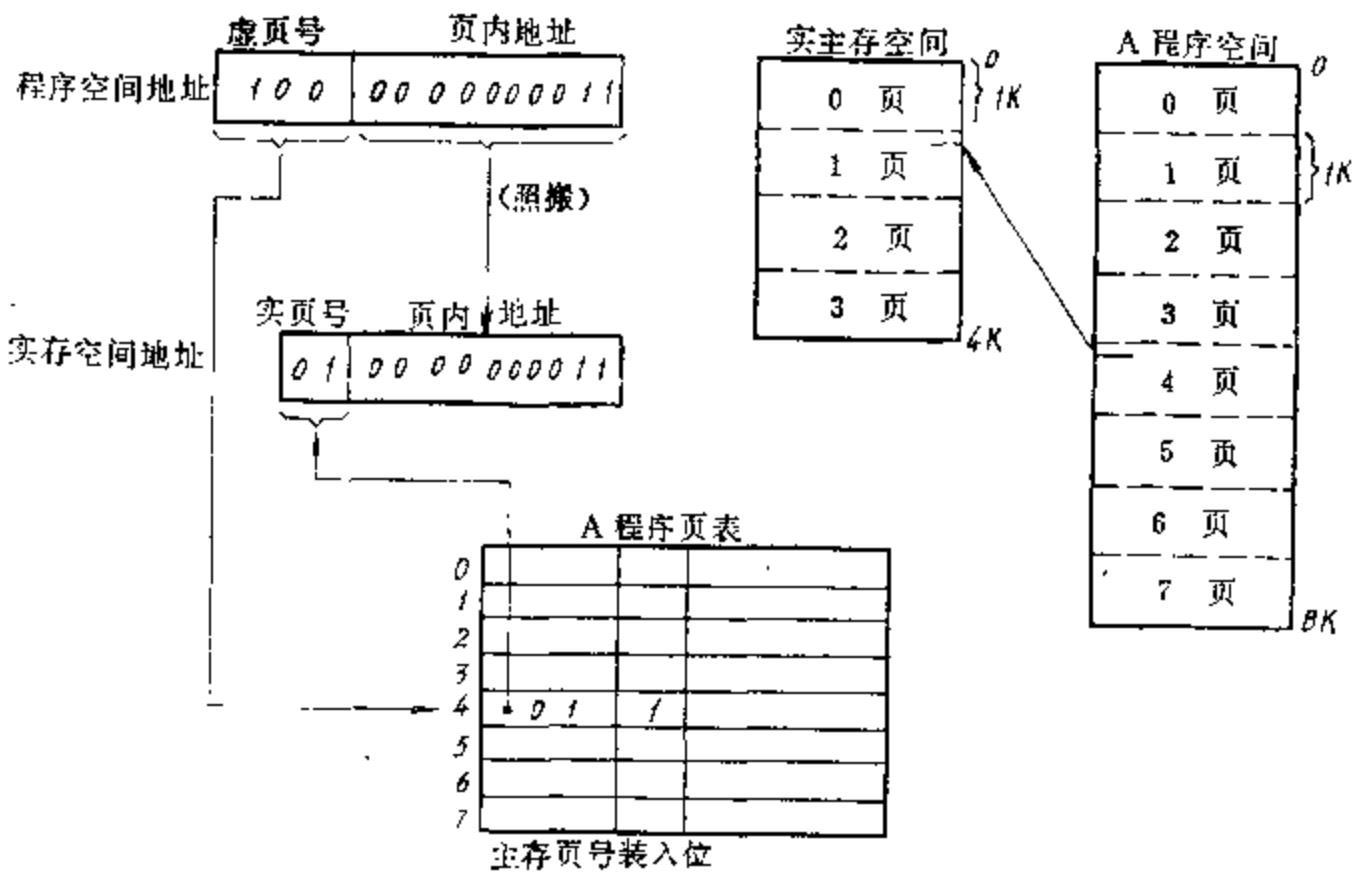


图 5.28 页面地址变换

的地址变换方式，所以这种页内另头是无法为其它程序所利用。显然，页面越小，这种页内另头的损失越小。与页内另头相应，把主存中各程序间的空白页称为外另头。

2.3-3 段式和页式的比较

用表格对比说明。

段 式	页 式
<ul style="list-style-type: none"> (1) 段的划分是按程序的逻辑结构; (2) 段的大小不定; (3) 程序的起点 (或段的起点) 原理上可以是任意的; (4) 段的终点要指明, 即段表中需有段长项; (5) 需要对主存中的空位与要进入的段的段长比较, 以判定是否有空位以及哪个空位能存得下要调入的段; (6) “外另头” 损失大; (7) 段表对应用程序员不是透明的; (8) 有助于程序员处理各个段之间的链接、可变长段、段的保护、段间的相互独立的修改和处理等; (9) 易于实现模块的公用, 这是其较大的优点。 	<ul style="list-style-type: none"> (1) 页面是实存、程序空间的机械等分; (2) 页面的大小固定; (3) 程序的起点必须是页面的起点; (4) 页的终点自明, 页表中不必有页长项; (5) 要把一个页调入主存, 只需查看主存使用表是否有空白页就有可能调入 (这和下节要讲的地址映象方式有关); (6) “内另头” 比段式的 “外另头” 小; (7) 页表对应用程序员是透明的; (8) 程序员编制程序时可不顾主存实际容量, 不必安排程序在主、辅存之间如何传送, 这是其很大优点; (9) 各页不是逻辑实体, 不便于实现程序和数据的公用及相应的保护。

总的来说, 由于页式管理的地址变换快得多, 调入操作也简单得多, 因此采用它的机器越来越多, 以致纯段式管理的机器几乎没有了。

2.3-4 段页式

从上面可以看出, 段式和页式各有其优、缺点, 为了获得这两者的优点, 弥补各自的缺点, 可以采用分段和分页的结合。把实存等分成页, 程序按模块分段, 每个段分成与实存页同样大小的页面。每道程序有一个段表和一组页表, 段表中每行对应一个段, 每行有一个指向该段的页表的起始地址及该段的控制保护信息, 由页表指明该段各页在主存中的位置和是否已装入、已修改等控制用信息。由于采用分页, 所以与纯段式的重要不同是段的起点不能是任意的, 必须是位于实存中页面的起点。

多道程序的每道 (每个用户) 需要有一个基号 (用户标志号), 可由它指明该道程序的段表起点存放在哪个基寄存器中。这样, 程序地址应包括基号 b 、段号 s 、页号 p 、页内地址 d 。格式如下

基 号 b	段 号 s	页 号 p	页内位移 d
---------	---------	---------	----------

现举例说明段页式的地址变换过程。

假设实主存分成 32 个页面, 有 A、B、C 三道程序已经占用主存中如图 5.29 所示的阴影部分, 现在又有 D 道程序要进入。它有三段, 段内页号分别为 0、1; 0、1; 0、1、2。

如果采用纯段式, 虽然主存有八个页空着, 超过 D 道程序需要的七个, 但因第 2 段有三页比任何空隙大而无法进入。采用段页式后则可调入, 各段各页在主存的位置如图 5.29 所示。

当要访问的程序地址是 D 道 2 段 1 页内的 d 时, 程序地址到实存地址的变换过程如图 5.30 所示。

先由基号 D 经基寄存器找到 D 道程序的段表的起点 S_D , 加上段号 2 找到 D 道 2 段的

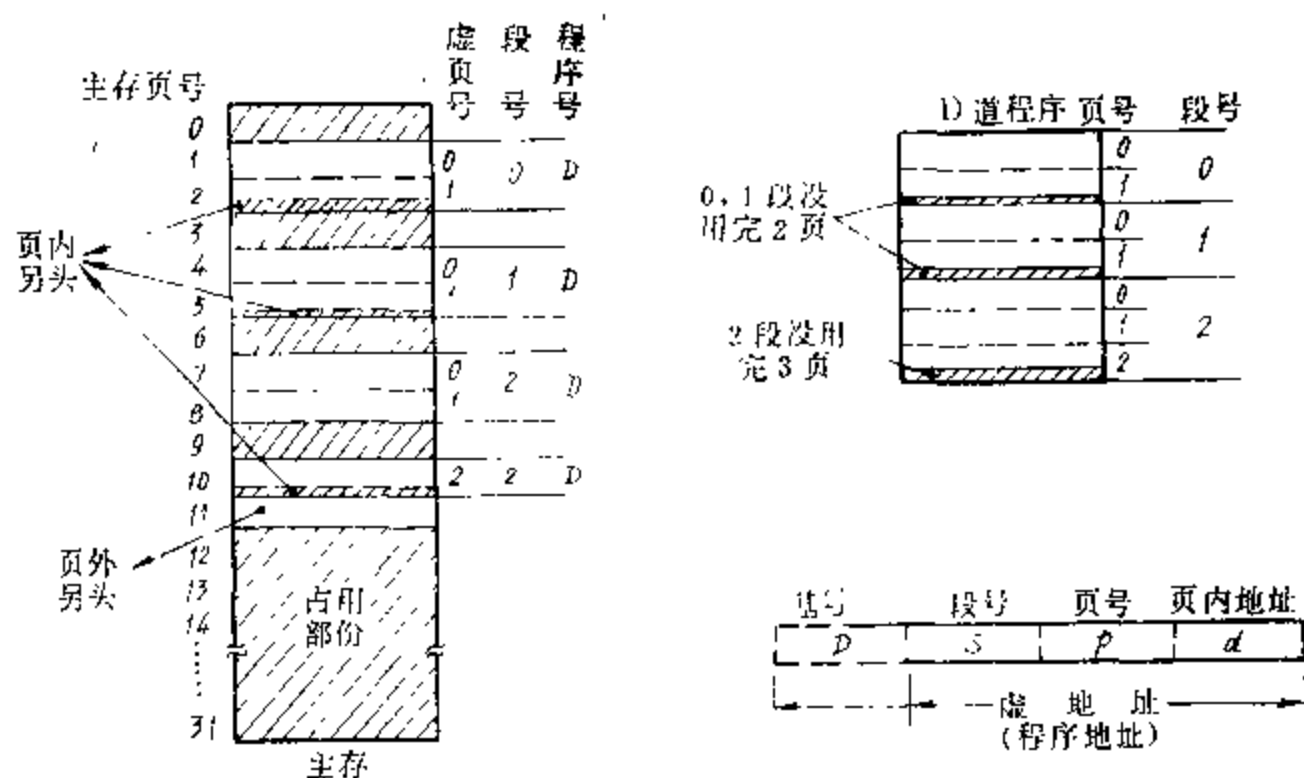


图 5.29 段页式存贮举例

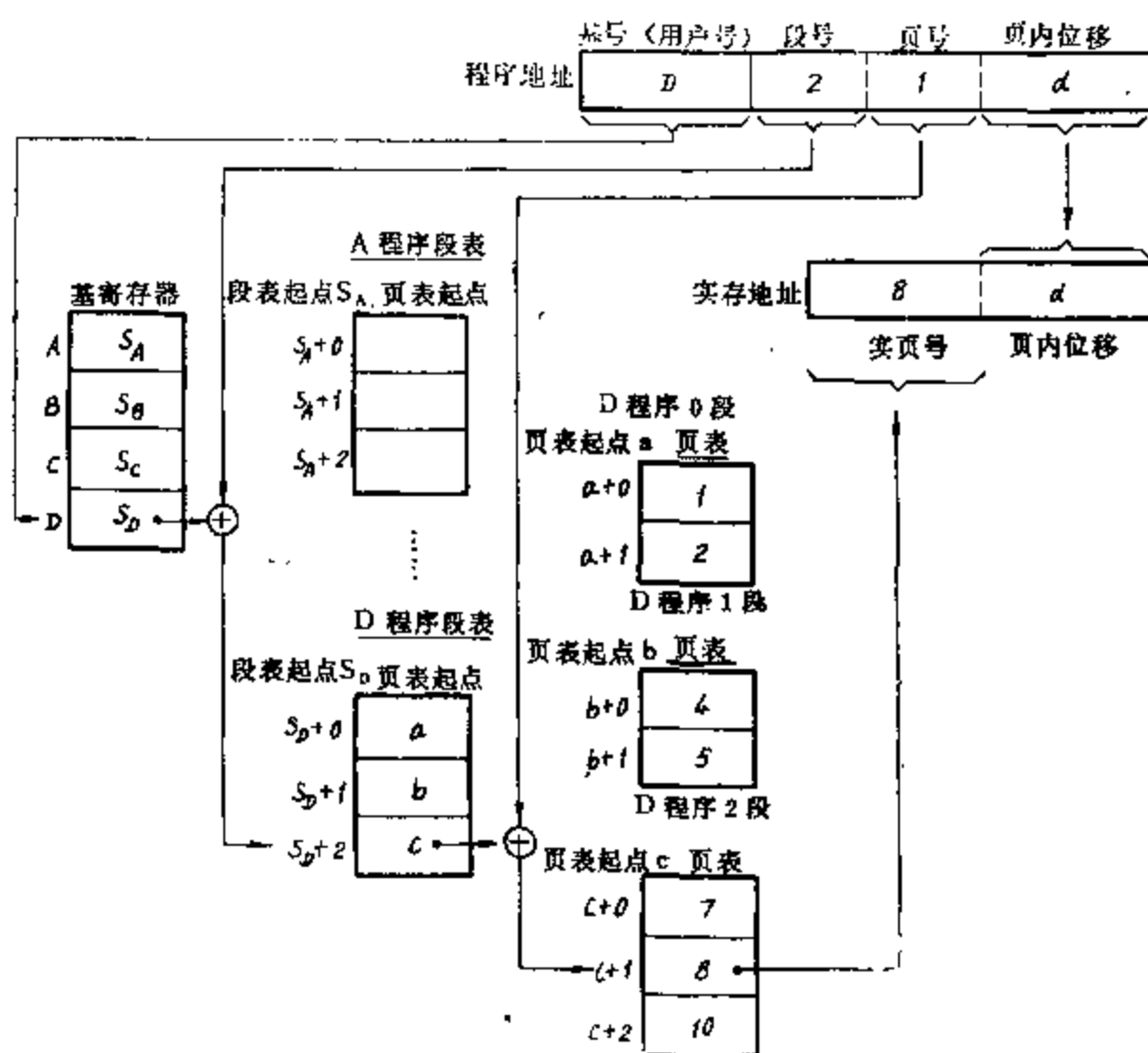


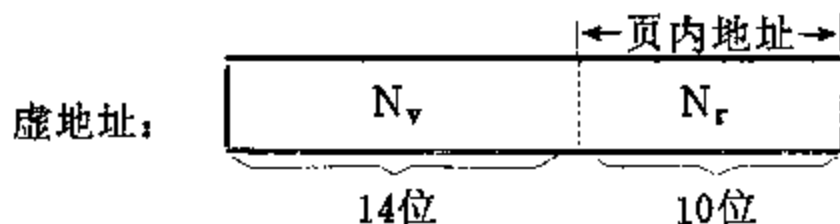
图 5.30 段页式地址变换

页表起点 C，再加上页号 1，找到对应 D 道 2 段 1 页的实存页号 8，最后拼接页内位移 d 即得该程序地址所对应的实主存地址。

目前，大中型机如 IBM370 等都采用这种段页式存贮。它既有页式的所有优点，又有段式的便于程序公用和保护的优点。缺点是地址的变换需多次查表，如果每次访存都需要如此查表，那是根本没法用的，在 § 5 要讲述解决的办法。当然，这里仍然存在页内另头。

看出，段页式由程序地址变换成实主存地址至少需查二次表（段表与页表）。段、页表

构成表层次。当然，表层次并不只是段页式有，页式也会有。这是因为前述的查表方法是基于整个页表是连续存贮，而当一个页表的大小超过了一个页面的大小，页表就可能分存于主存中不连续的各页内。这样，前述由页表起点加页号查得该页在页表中的行的查表方法就会出错，而页表大于页面是完全可能的，例如，若虚地址为24位，其中页内地址为10位，即



则页表就需要 2^{N_v} ，即 2^{14} 行，而页面大小为 $2^{N_r} = 2^{10}$ ，因此页表本身就可能要分存于16个页面。使每个页表大小限制在一页之内，是采用表层次的目的之一。

用树的概念可以容易地得出层次数 i 和 N_v 、 N_r 的关系式：

$$2^{N_v} = (2^{N_r})^i$$

$$i = \frac{\log_2 2^{N_v}}{\log_2 2^{N_r}} = \frac{N_v}{N_r}$$

因为 i 必须是整数，所以向上取整得 $i = \left\lceil \frac{N_v}{N_r} \right\rceil$

对于上例来说 $\left\lceil \frac{14}{10} \right\rceil = 2$ ，即需要二级页表。从这种意义上讲，也可把段页式看成是为了弥补纯页式中页表行数超过页面大小而增加的层次。如果表中的一个项目需要 B_e 个编址单元，若 B_e 为2的幂数，则 B_e 需用 $N_e = \log_2 B_e$ 地址位表示，这样就有

$$i = \lceil N_v / (N_r - N_e) \rceil。$$

例如Multics 其 $N_v = 26$ ， $N_e = 1$ （表中一项需二个字节）， $N_r = 10$ ，故表的级数为 $\lceil 26 / (10 - 1) \rceil = 3$ ，就是说，只分成段页二级还不够。

还要认识到，采用表层次技术并不能减少所需的页表（第二级表）空间，甚至还要多用了段表（第一级表）空间；然而好处在于在程序运行时除了第一级表需驻留在主存之外，整个页表中只需有一部分是在主存内，大部分可存在辅存，在需用到时再由第一级表调入主存，从而可减少每道程序所占用主存空间。这也可看成是存贮层次概念的一种应用。VAX-11/780 对用户来讲，虚地址中并没有段号，然而由于上述的原因，在地址变换中采用了二级表层次。

§ 3 地址的映象与变换

虚拟存贮体系中，把虚存（辅存）空间划分成 2^{N_v} 个大小为 2^{N_r} 编址单元的页面，但实存（主存）空间只有与虚页同样大小的 2^{N_v} 个页面。那么虚地址如何变成实地址呢？所谓的地址的映象和变换就是讨论这个问题的。前面讲过，规定程序的始点必须是页面的始点，使得虚页页内地址 N_r 能直接原样地成为实主存页内地址 n_r ，即这部分地址是不必变换和映象的；因此，地址的变换就是 N_v 如何变成 n_v ，见图 5.31。

所谓地址的映象是指每个虚页按什么规则装入实存，即虚地址 N_s 与实地址 n_s 之间如何

对应。地址的变换则是指的当程序按照这种映象关系装入实存后，解题时，程序(虚)地址 N ，如何变换成对应的实存地址 n_p 。地址的映象和变换是密切相关的，所以我们结合起来讲述。

一般地， $2^{n_v} \ll 2^{N_v}$ ，因此地址的映象实质上是怎样把一个大的程序空间压缩、映象到小的实存空间。这样，实存中每一个页面必须能与多个虚页相对应。能对应多少个虚页是随映

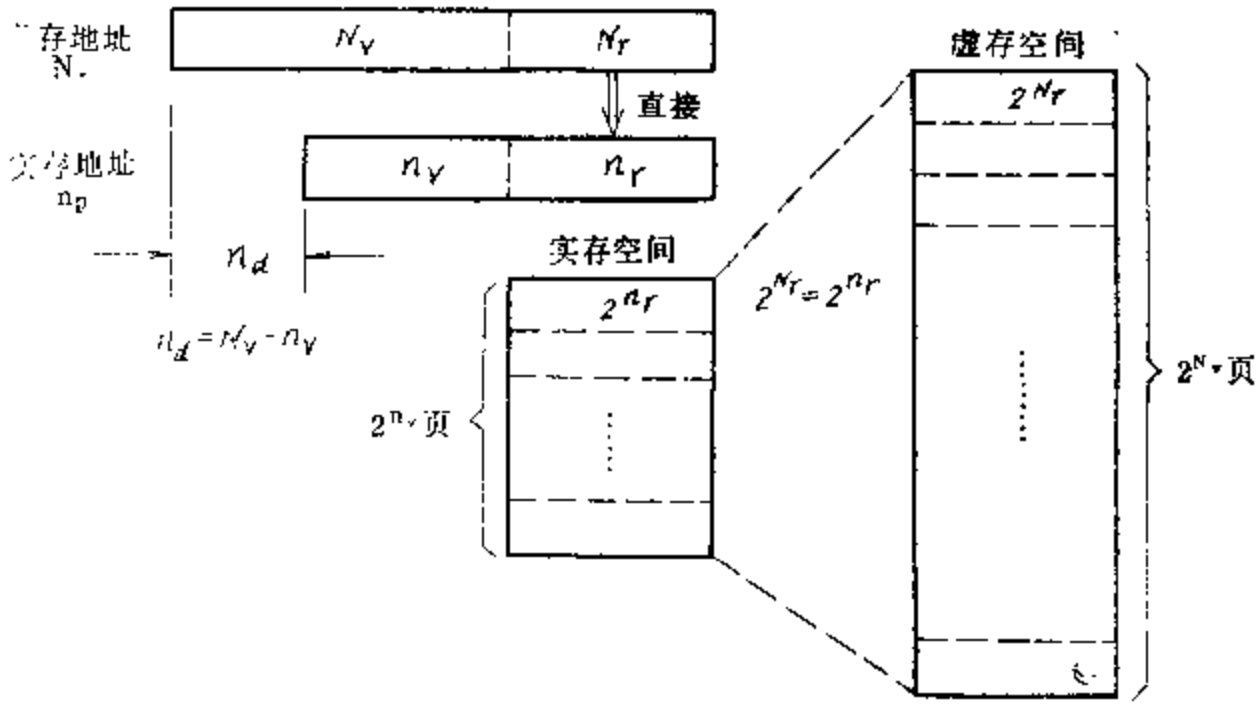


图 5.31 虚存空间压缩成实存空间

象方式而异的，但至少应是虚存总的页面数除以实存总的页面数，即 $2^{N_v}/2^{n_v} = 2^{n_d}$ 。当同时有二个以上的虚页要进入同一个实页位置时，就会产生实页冲突或页面争用，它是选择映象方式时要考虑的一个重要因素。

地址的映象有多种方式，下面分别讨论四种不同的映象。

3.1 全相联映象及其变换

任何虚页能映象到实存任何页面位置的方式称为全相联映象，如图5.32。从地址来看，就是任何 N_v 可对应于任何 n_v 。只有当一个题目要求同时调进的页数超出 2^{n_v} 个页面（这是很少见的）时，二个虚页才可能争用一个实页位置。因此，全相联映象的实页冲突概率最小，是理想的映象方式。

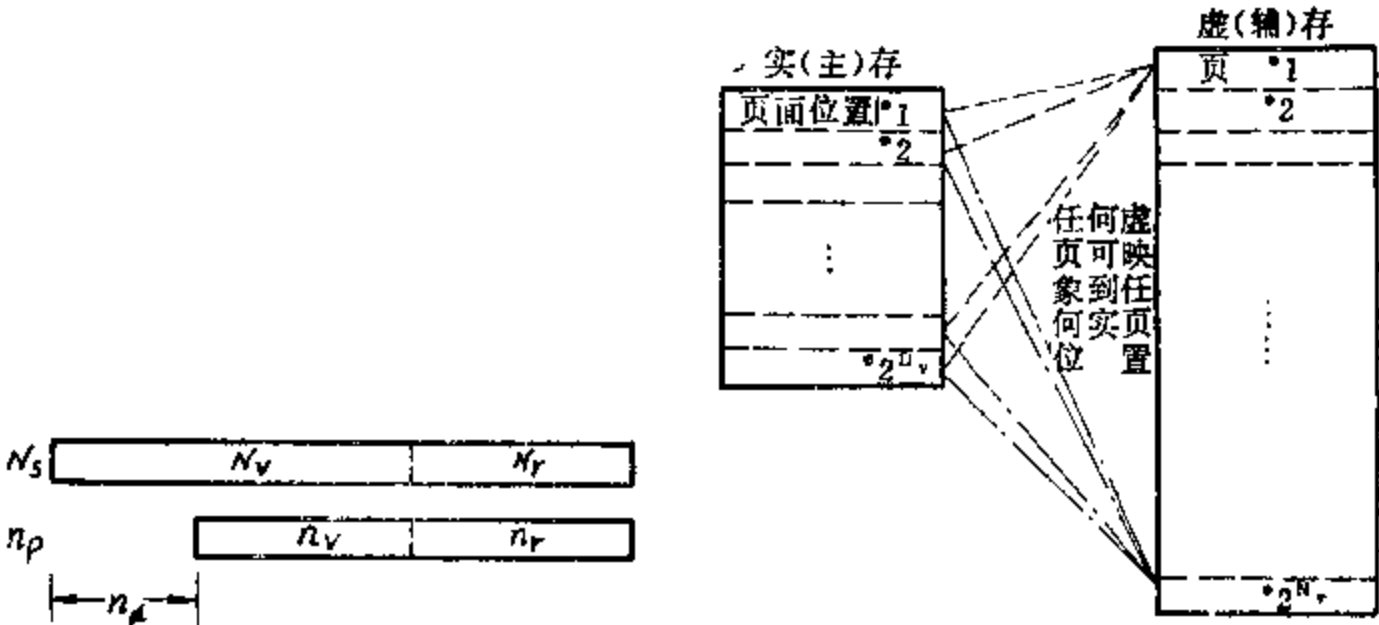


图 5.32 全相联映象

对这种映象，其直观的地址变换方法是页表法。例如，设 N_v 为4位， n_v 为2位，如图5.33所示，若 $N_v = 0100$ ，则查页表第0100行，其装入位为1，表示该页已装入实存中，并查得该页存在第11（二进制）实页处，实页号与 N_r 拼接即得对应的实存地址。这种页表可存在随机访问存储器中，共需 2^{N_v} 个入口。由 N_v 查得页表的对应入口，若装入位为0时，表示该虚页未装入实存，即出现页面故障，这时就需要由辅存把该页调入。

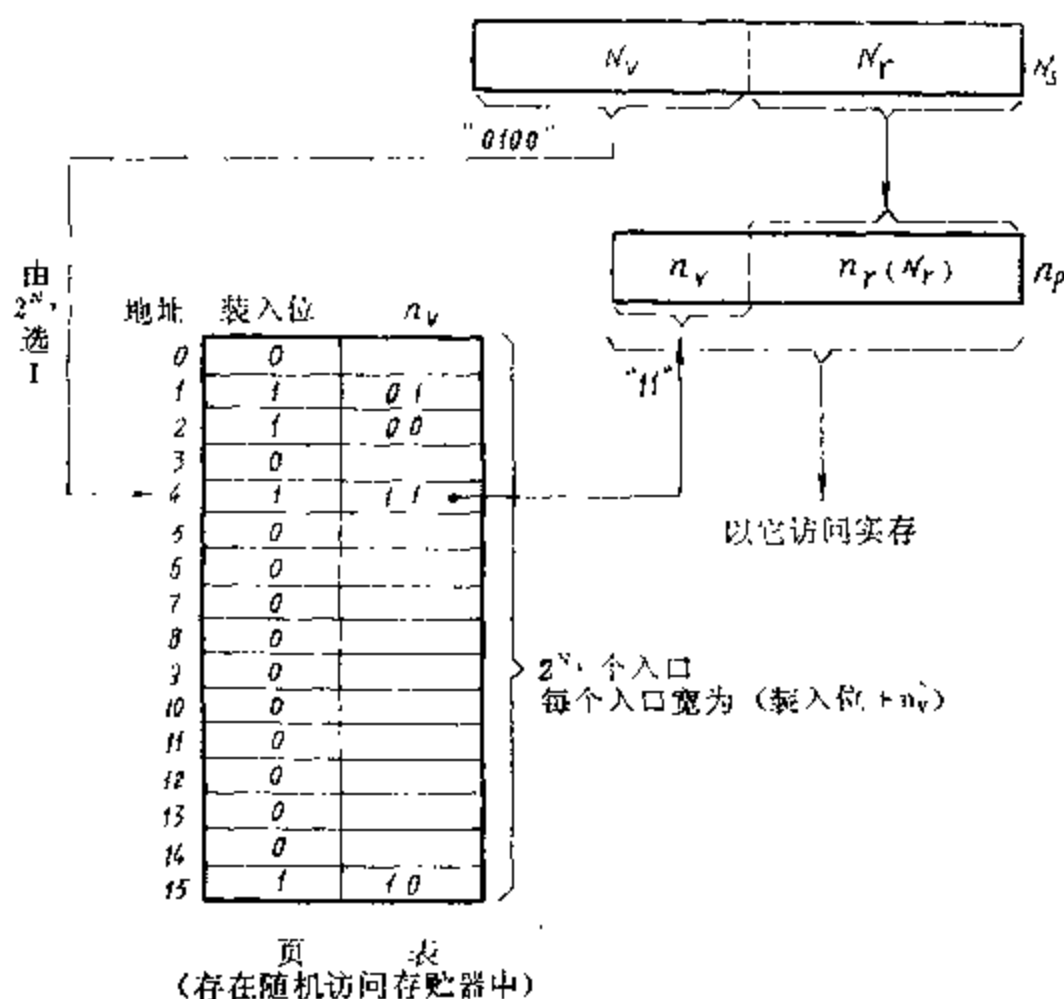


图 5.33 查页表法

页表中装入位为“1”的入口最多只有 2^{n_v} 行（本例为四行），这是因为实存中只有 2^{n_v} 个实页。既然其它入口的装入位都是“0”， n_v 项没用，那么能否把页表压缩成只是由已装入的入口构成呢？对于上例来说就是按图5.34那样构成。显然，它不能用按地址访问的存储器来存贮，而是用按内容 N_v 查找 n_v 的相联存储器来存贮。它不需要装入位，如果程序地址的 N_v 在相联存储器中查到，表示该虚页已被装入实存，如相联查找不到就表示该页未装入。这种方法称为相联目录表法，或目录表法，它需 2^{N_v} 个入口，每个入口为 $N_v + n_v$ 位宽。

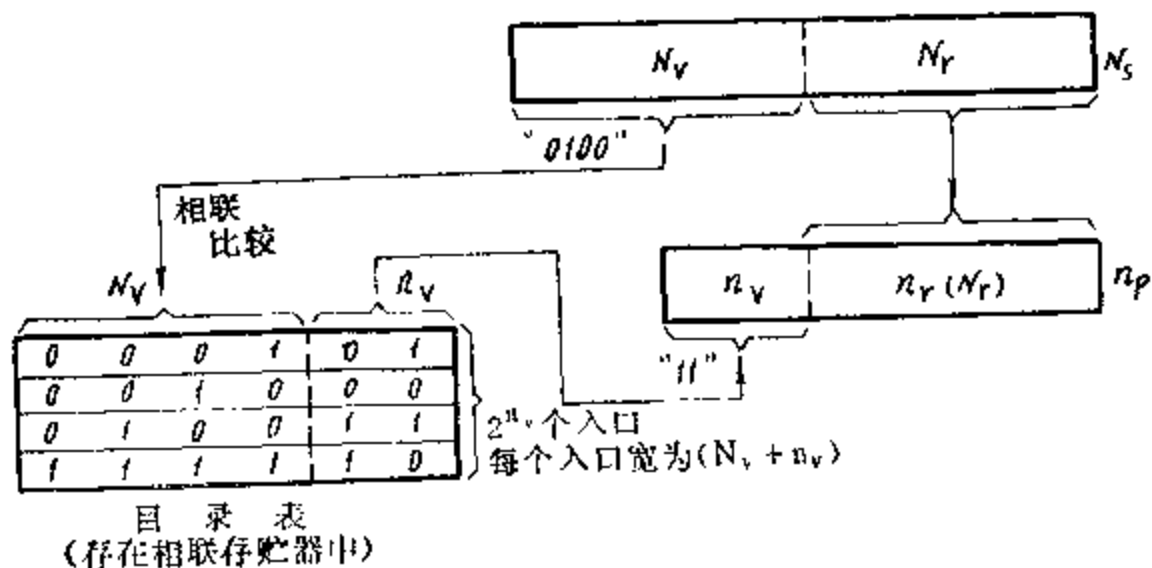


图 5.34 目录表法

前面讲过，页表所需容量可能大到需采用表层次，查表速度慢。页表可存在通用存贮器内，而目录表只能是存在专用的存贮器内。虽然目录表所需容量可以较小，但由于需采用相联存贮技术，要构成 2^{n_v} 容量的相联存贮器，造价会很高，而且查表速度也难以很快。那么，能否缩小这二种表的容量呢？能否有可缩小表格的其它映象方式呢？下面讲的直接映象就是一种。

3.2 直接映象及其变换

每个虚页只能映象到实存一个特定页面的方式称为直接映象。如图 5.35，虚存第 1 页，

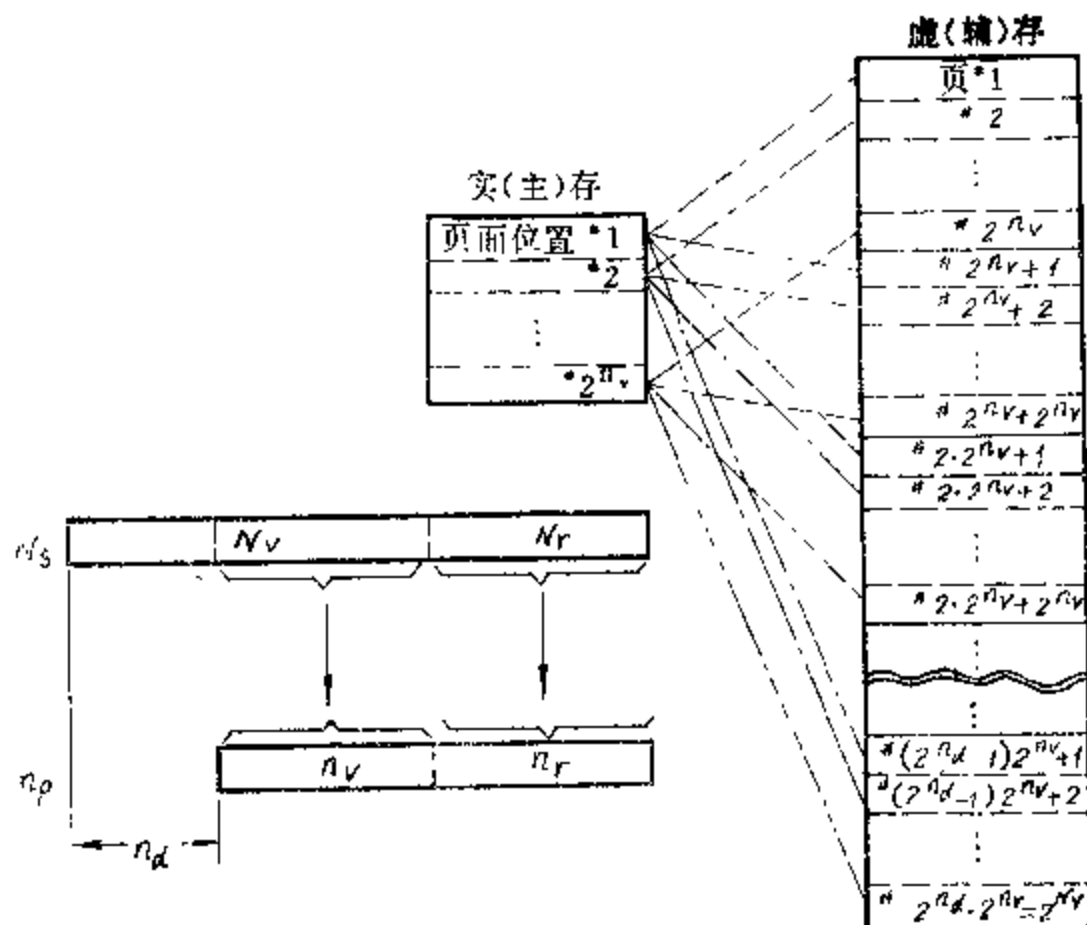


图 5.35 直接映象

第 $2^{n_v}+1$ 页，第 $2 \cdot 2^{n_v}+1$ 页，……，直至第 $(2^{n_d}-1) \cdot 2^{n_v}+1$ 页只能映象到实存第 1 页；而虚存第 2 页，第 $2^{n_v}+2$ 页，第 $2 \cdot 2^{n_v}+2$ 页，……，第 $(2^{n_d}-1) \cdot 2^{n_v}+2$ 页只能映象到实存第 2 页；如此等等。这相当于把程序空间按实存空间分块，而块内各页直接映象到实存的相应页。

直接映象的好处是由 N_v 变化到 n_v 只需将 N_v 中与 n_v 对应的部分与 N_r 拼接即可。当然，用此拼接地址访问实存时，还需要判断这个虚页是否已在实存中，因为能映象到这个实页位置的有 2^{n_d} 个虚页，而当前已在该实页位置的虚页并不一定正好是要访问的那个虚页，这种判断要用到 N_v 中对应的 n_d 部分。

一种方法是用 n_v 去查对应的 n_d 看是否是该虚地址的 n_d 值，如果相符，表示该虚页已在实存中，否则，产生页面失效（故障），见图 5.36。

此法的优点是用 n_p 去访实存可与判 n_d 是否相符的查表同时进行，如果判断出访问的虚页不在实存中，则按 n_p 进行的访问作废，这节省了时间，不必如前述查表法那样先查表后访问实存。

直接映象的致命弱点是实页冲突概率高。举例来说,要计算阵列

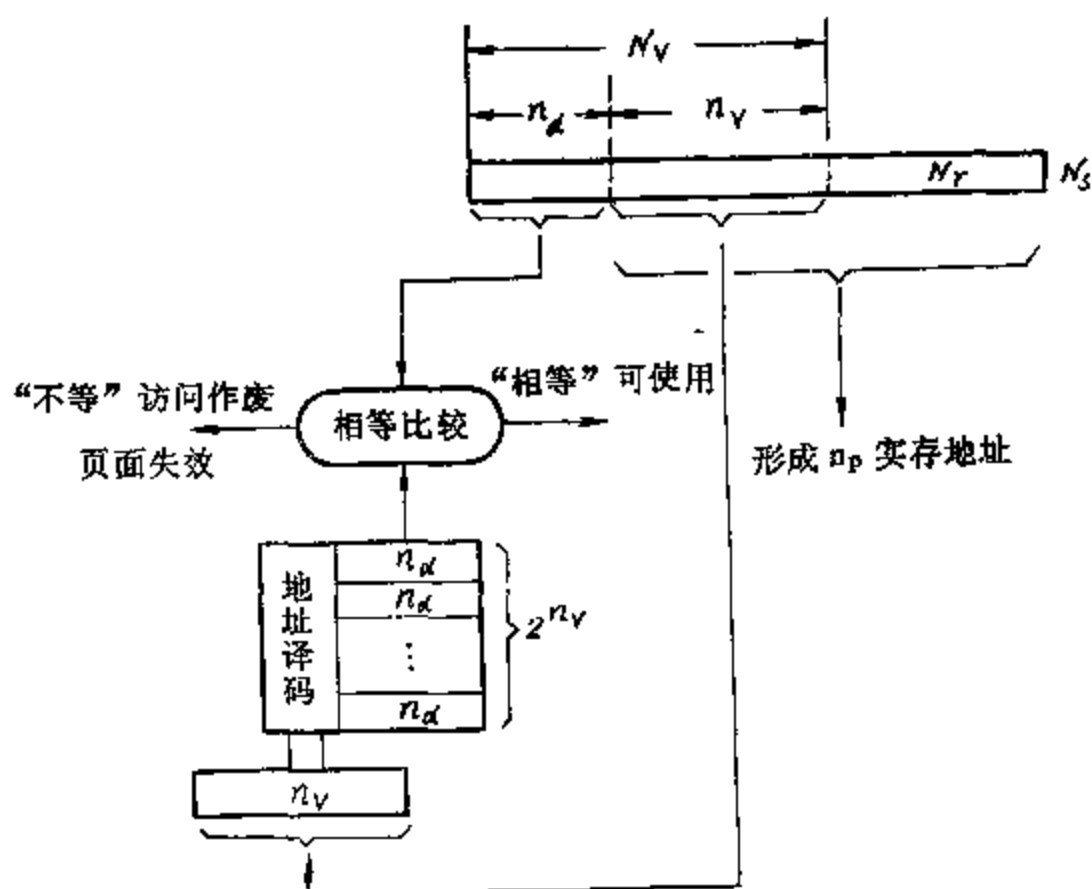


图 5.36 直接映象方式实现地址变换的一种方法

$$F_{ijk} = \sum (C_1 A_{ijk} + C_2 M_{ijk})$$

若 A_{ijk} 阵列编址在程序空间的第 m 页, 只要 M_{ijk} 阵列编址在 $L \cdot 2^{n_v} + m$ 页 (m 在 1 到 2^{n_r} 之间, L 在 1 到 $2^{n_d} - 1$ 之间), 就要发生实页冲突, 这是因为计算时是 A_{ijk} 和 M_{ijk} 同时要, 但当把 A_{ijk} 调入主存第 m 页后, 只能进入主存第 m 页的 M_{ijk} 就调不进来了。其次是直接映象还要降低实存的利用率, 当实存第 m 页没有空时, 即使实存其它页, 如第 $m+1$ 页有空时也无法装入。因此, 直接映象在主—辅层次中一般都不采用, 而只在个别小型机的主—Cache 层次中为省设备才有采用。

如果在直接映象的基础上加以改进, 将全相联映象和直接映象相结合, 就可以既能减少实页冲突概率, 又使地址变换比全相联映象的简单, 这就是下面要讲的组相联。

3.3 组相联映象及其变换

以简例来说明这种方式 (实际应用中地址码字段要长得多)。如图 5.37 所示, 将实存空间和虚存空间都分成组, 每组为 S 页 ($S = 2^s$)。实存共 2^{n_v} 个页, 分成 Q 组 ($Q = 2^q$), 整个实存是一块。虚存分成与实存同样大小的 2^{n_d} 块。虚存地址按块、组、页号分成对应的若干字段。虚地址的组号、组内页号分别用 q' 、 s' 字段表示, 它们的宽度和位置与实地址的 q 、 s 是一致的。

组相联映象指的是各组之间是直接映象, 但组内各页间则是全相联映象。图中, n_d 、 q 都是 1 位, s 是 2 位, 即虚存的第 0 组只能进入实存 0 组, 而第 1 组只能进入实存 1 组, 组内的各个页, 如虚存的 0、1、2、3 及 8、9、10、11 页可进入实存 0、1、2、3 中任意一页, 但不能进入实存 4、5、6、7 页。看出, 组相联映象是介于“全相联”映象与“直接”映象之间。它的实页冲突概率要比“直接”的低得多, 例如, 当虚存 0 页已在实存 0 页中, 若要调入虚存第 8 页则对直接映象来说就要发生实页冲突, 而对组相联映象来说, 第 8 页仍可进入实存

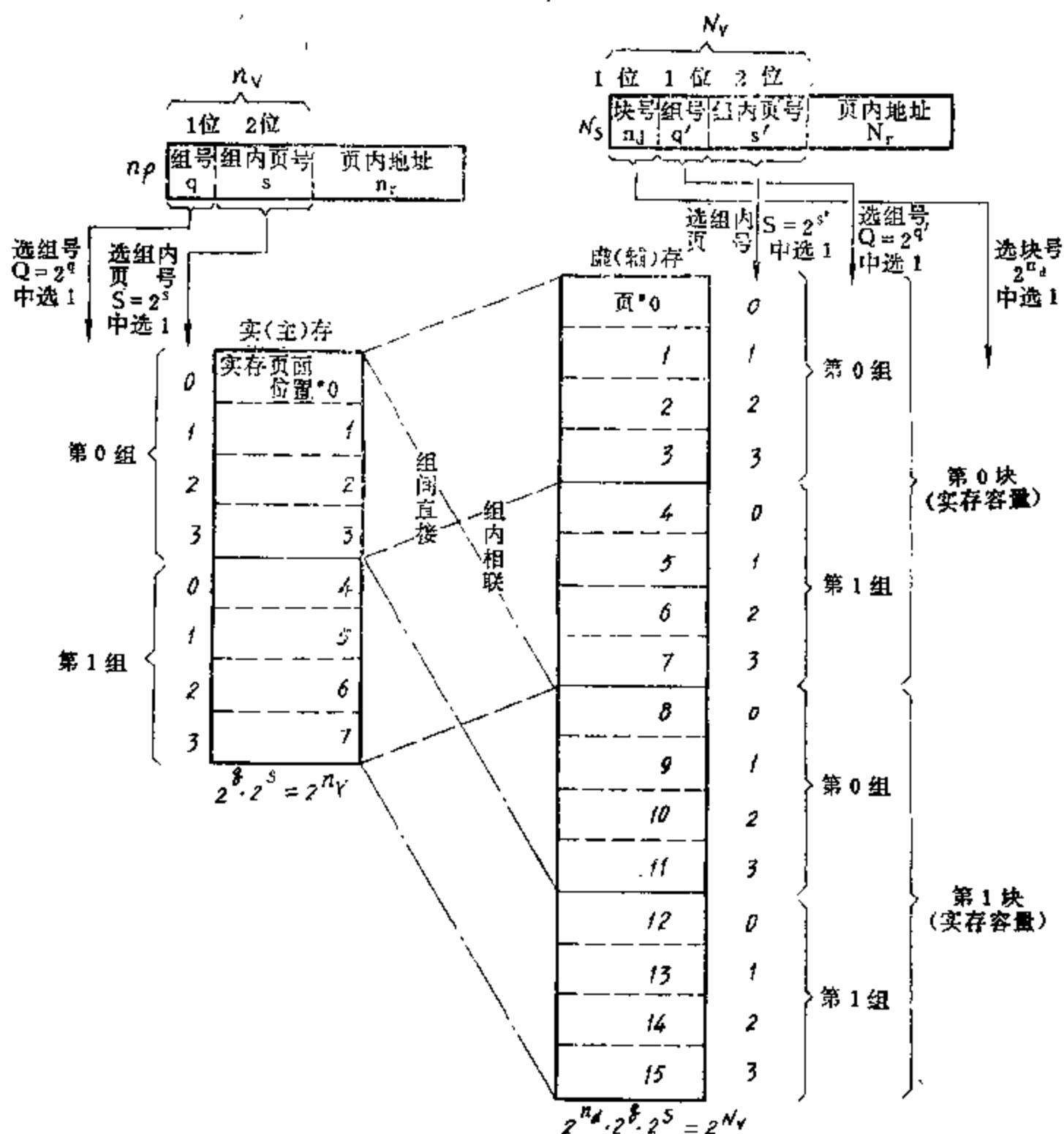


图 5.37 组相联映象

1、2、3 中任意一个页面位置。显然，1、2、3 页此时都被占用的概率要小得多，从而大大降低了实页冲突的概率。S 值越大，“实页冲突”的概率就越低。

组相联映象，当 S 值大到等于实存页面数 $S = 2^{n_v}$ （即 $s = n_v$ ）时就变成了全相联映象；而当 S 值小到只有 1 页（即 $S = 2^0 = 1$ ）时就变成了直接映象。因此全相联映象和直接映象只是组相联映象的二个极端。在实存空间大小及页面大小已定的情况下， $(s + q)$ 的位数就定死了，但“存贮体系”的设计者仍可对 S 和 Q 值进行选择。S 值愈大，实页冲突概率愈低，但地址变换愈复杂；S 值愈小，地址变换愈易实现，但会使实页冲突概率上升。对于主—Cache 层次，一般认为取 $S = 2 \sim 4$ ，就会使实页冲突概率明显下降。

现在来看组相联映象的地址变换过程。

前面在全相联中讲过的目录表法可用于实现组内的全相联，但目录表的容量可从全相联的 2^{n_v} 减少到 2^s 。因为各组间是直接映象，所以 q' 可照搬到 q ，且 q' 不必参加相联比较，目录表的键符从全相联的 N_v 减少到 $N_v - q' = n_d + s'$ 位，这会提高查表速度。

这样，组相联的地址变换原理如图 5.38 所示。先由 q' 在 2^q 中选出一组，对该组再用

$n_d + s'$ 进行相联查找，若在 2^s 行中查不到相符的，则表示该虚页不在实存中；如果查得到相符的，则将表中相应的 s 拼接上 q' 就是实存地址的 n_v 。表的总行数应为 $2^q \cdot 2^s = 2^{n_v}$ 。

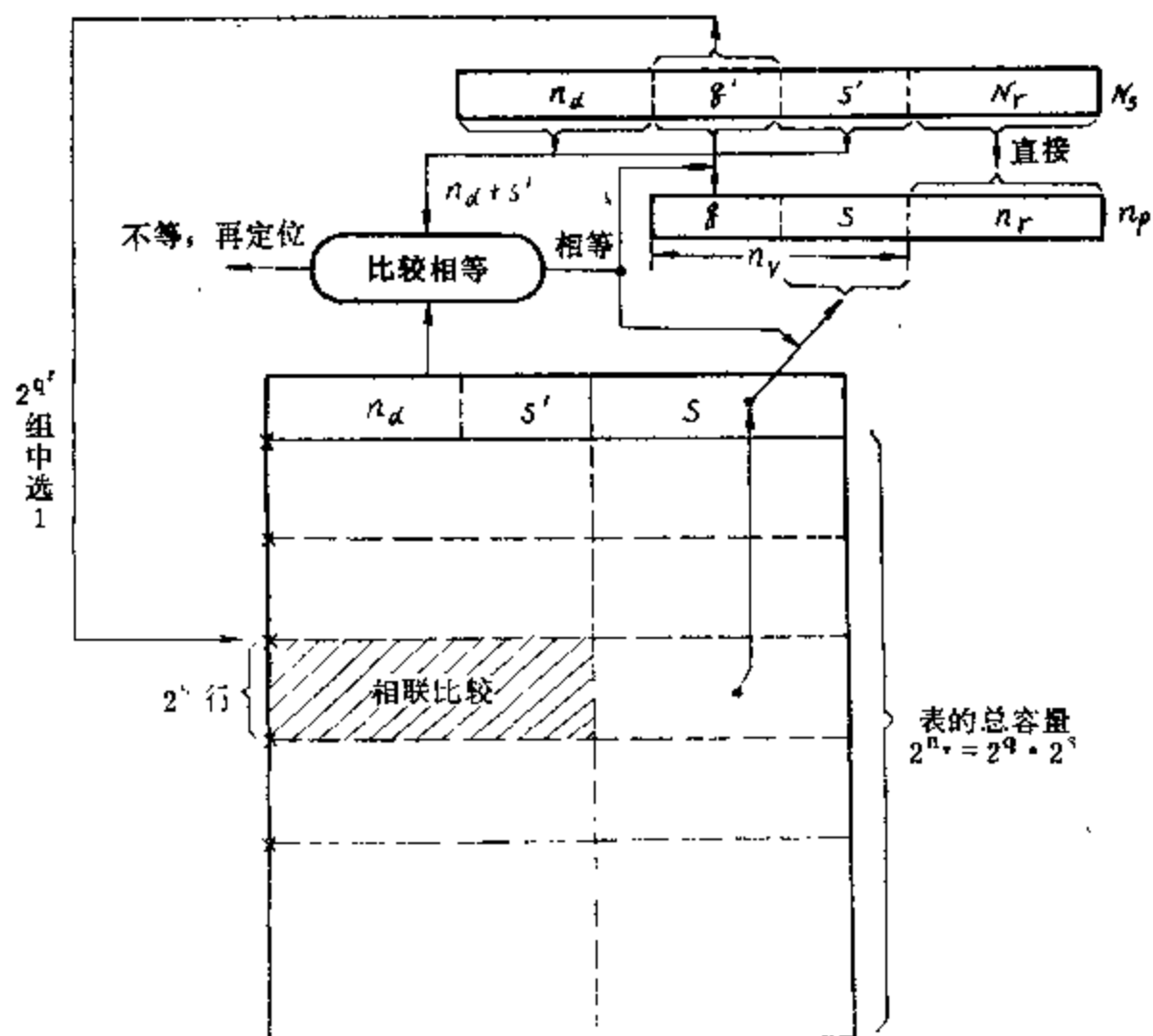


图 5.38 组相联地址变换示意图

这个表既要相联比较又有直接寻址，那就需用按地址访问与按内容访问混合的存储器。

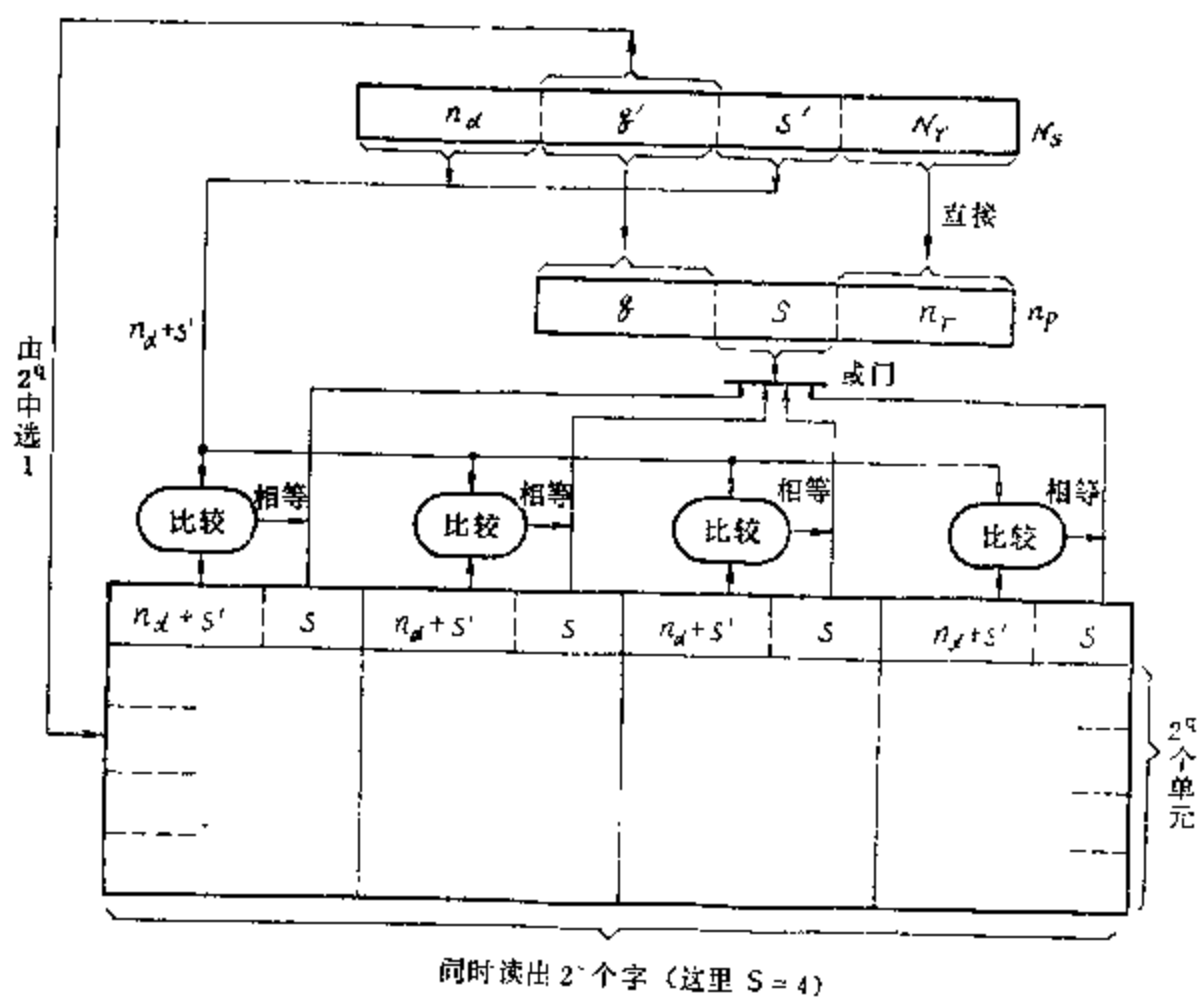


图 5.39 组相联地址变换的一种实现方式

在七十年代，相联存贮片子的速度满足不了要求，那就只能用速度快得多的随机访问存贮器片子来构成按内容访问与按地址访问相混的表。

一种办法是使用 § 1.3 节中讲过的单体多字并行存贮器，如图 5.39 所示。先由 q' 从 2^q 中选出一个单元，由该单元同时读出 2^s 个字，分别与虚地址的 $n_d + s'$ 通过 2^s 套外比较电路同时比较。将其中比较符合的 s 取出拼接上 q' 即为实存页号 n_v 。如果都不相符，表示该虚页不在实存中。显然，只有当 S 值很小（如图上的 $S = 4$ ）时，才能用这种方法。

采用组相联并不是操作系统或存贮层次的要求，只是在全相联的速度满足不了要求时才采用，它是在便于实现和减少实页冲突概率之间进行取舍。随着半导体集成电路技术的发展， 2^s 值还可增大，以进一步降低实页冲突概率。

3.4 段相联映象

以上讨论了全相联、直接、组相联这三种映象。在这三种映象的基础上还可以有各种变形，段相联就是一例。所谓段相联实质上是全相联的一个特殊情况，它是段间全相联，但段内各页是直接映象，如图 5.40 所示。

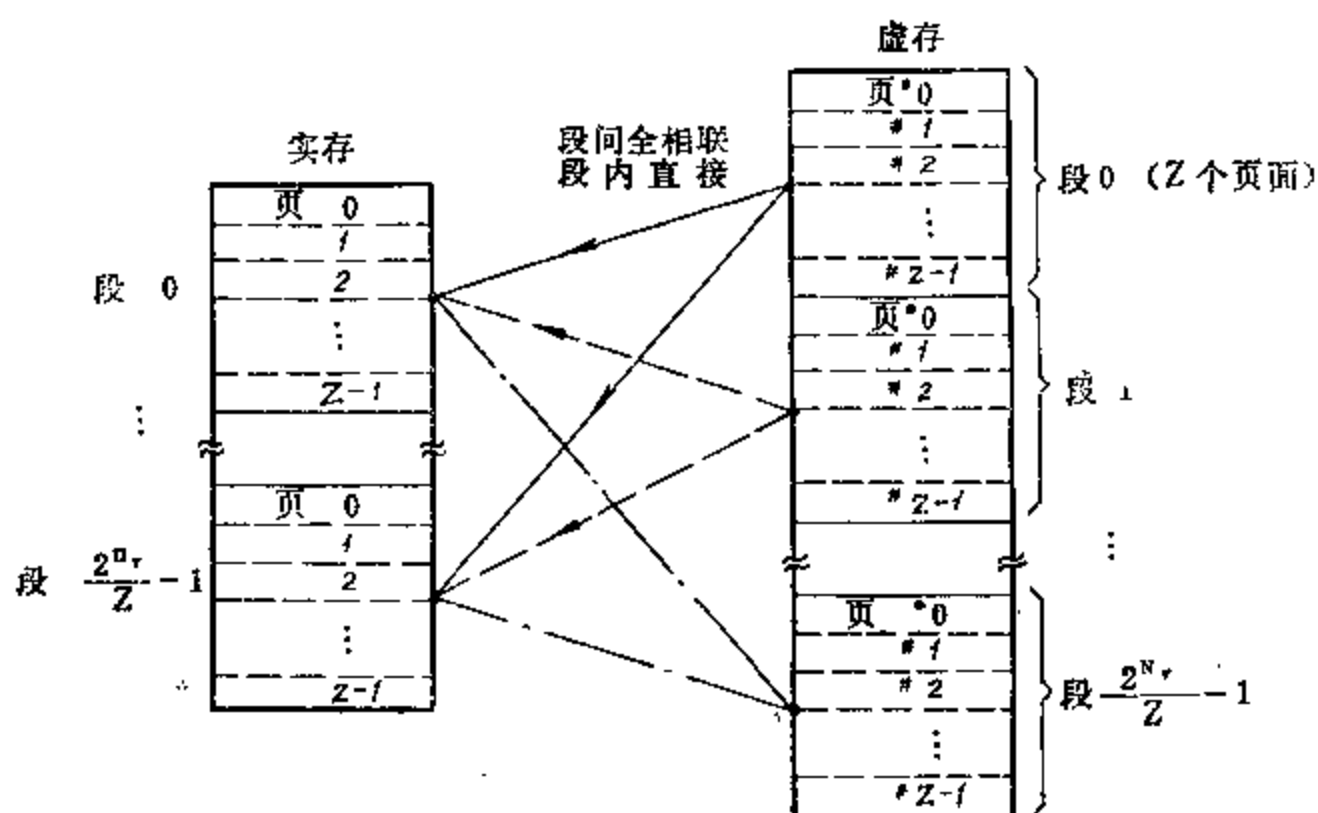


图 5.40 具有每段 Z 个页的段相联一般形式

采用段相联的目的在于减小相联目录表的容量，使之从 2^{n_v} 减为 $2^{n_v}/Z$ ，（若 $Z = 16$ ，所需相联存贮器容量减少到 $1/16$ ）。但是，这也使虚存的每段与实存内能全相联的段数只有 $2^{n_v}/Z$ ，从而会使实页冲突概率增加。

3.5 对标志表的分析

从上述各种地址映象和变换中可以看出，为了实现地址变换都需要有各种映象“表”。这些表统称为标志表。

为了比较，表 5.5 中列出了全相联查表法、全相联目录表法、组相联目录表法的标志表特性。

当然我们不能仅从表的总容量来判断好坏，因为相同容量的相联存贮器要比随机访问存

贮器成本高而速度低。因此要根据所用硬件的速度、容量、成本，以及对存贮体系的要求，如页冲突的概率等综合考虑。但是，就全相联映象查表法与组相联映象目录表法比较，后者的造价一般要低得多。

表 5.5 各种标志表的特性比较

		全相联查表法	全相联目录表法	组 相 联
行数 (入口数)		2^{N_v}	2^{N_v}	2^{N_v}
入口的状况	直接访问	2^{N_v}	0	2^s
	相联访问	0	2^{N_v}	2^s
相联比较的位数		0	N_v	$n_d + s$
每个入口的最窄宽度 (注)		$n_v + 1$	$N_v + n_v$	$n_d + 2s$
最小存贮容量		$(n_v + 1) \cdot 2^{N_v}$	$(N_v + n_v) \cdot 2^{N_v}$	$(n_d + 2s) \cdot 2^{N_v}$
可以采用的存贮器硬件		按地址访问的单体 单 字 存 贮 器	按内容访问的相联存贮器	按地址访问加上按内容访问的存贮器。S 值小时用按地址访问的单体多字存贮器

(注) 实际上还需要加某些控制信息位。

我们在第二章已讲过 Huffman 压缩概念。从压缩概念来看，上述的各种地址映象实质上是如何把一个大的程序空间“压缩”到小的实存空间中去；而且地址变换中，由查表法到目录表法也是一种压缩。用压缩概念来理解存贮层次的地址映象和变换很有好处，这便于如何从理论上来探索效率更高的映象方式。

3.6 散列 (Hashing) 概念在地址变换中的应用

上面说过，全相联目录表要求能按 N_v 内容访问，因此就得用相联存贮器片子构成，但相联存贮器片子价格高、容量小、速度慢，那么，能否用按地址访问存贮器来构成呢？当需相联比较的行数很少时，是可以单体多字按地址访问存贮器来构成，如图 5.39 所示。但对于容量达 2^{N_v} 的全相联目录表，是不能用读多字同时比较的方法的。

我们在“数据结构”课中已经学过，对在按地址访问存贮器中的信息实现按内容查找，可以有顺序查找、对分查找和散列查找等多种方法，其中以散列方法的速度最快。散列方法的基本思想是让内容（也称键符 Key）与存放该内容的地址 A 有某种散列函数关系，即 $A = H(\text{Key})$ 。存入时，把 Key 和其余内容存入存贮器的 $A = H(\text{Key})$ 单元中；查找时，将给定的 Key 经散列函数变换成 A 后，按地址 A 去访存，就有可能找到存贮器中存放该 Key 所在单元的其余内容。如图 5.41 所示。

散列函数有很多种。例如：

(1) 相除法

将键符 Key 除以一个常数后，或者取其商数 Q，或者取其余数 R 作为 A。相比之下，

采用余数法较好。因为，如果除数为 C ，余数 R 必在 $0 \sim (C-1)$ 的范围之内，即散列后的地

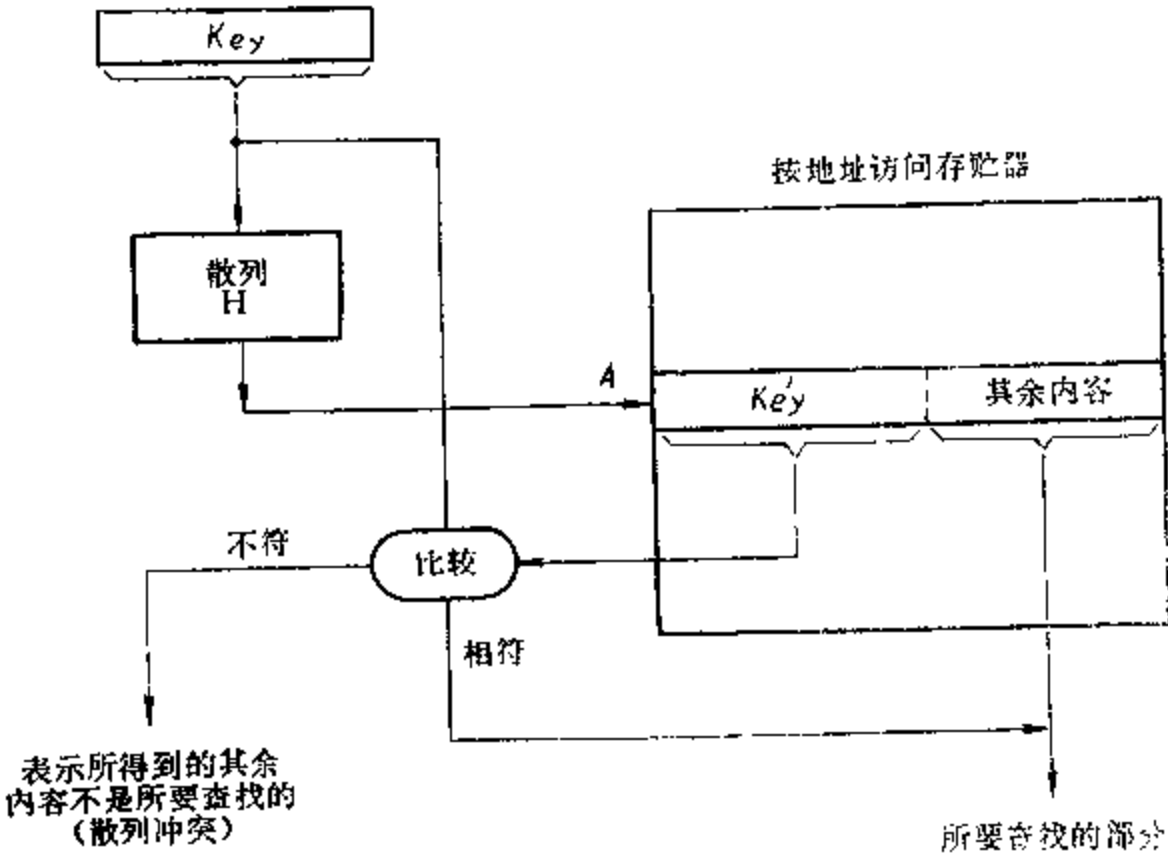


图 5.41 散列查找示意图

址必在 $0 \sim (C-1)$ 的范围之内。而且，可以证明如果 C 为质数，所得的余数可以较均匀地分布于 $0 \sim (C-1)$ 之间。因为同余数的数肯定不止一个，例如取 $C = 31$ ，对 Key 为 96、127 来说，对应 $Key_1 = 96$ 的地址 $A_1 = H(96) = 3$ ，对应 $Key_2 = 127$ 的地址 $A_2 = H(127) = 3$ ，两者一样，即 $A = H(Key_1) = H(Key_2)$ 。这种多个键符对应同一个地址的现象称为散列冲突，它是散列法所不能避免的。这就是说，对散列法，键符和地址不是唯一的对应的。

(2) 相乘法

将键符乘以某个常数后，取乘积中的某一段作为散列地址 A ，如图 5.42 所示。这同样也会有散列冲突。

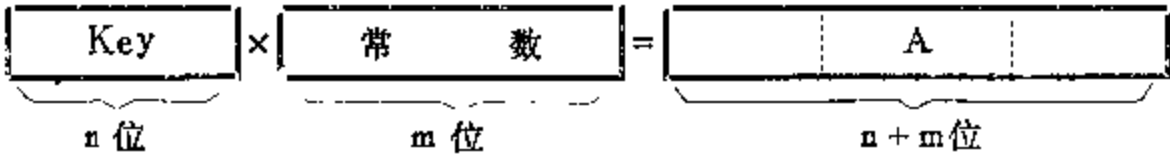


图 5.42 相乘散列法：A 的位数小于 n 位

(3) 折迭法

将键符按 q （应是质数）分成几段，不足 q 的高位补以零，然后对这些段进行某种处理，例如各段相加（或循环进位，或舍去进位）或按位加，以得到 q 位宽的散列地址 A 。显然，这同样也会出现散列冲突。

因此，在进行散列查找时还必须进行判别是否出现散列冲突，如图 5.41 那样，在把 A 中内容读出后，要将其 Key' 与给定的 Key 比较。若两者相符，则其余内容就是所要查找的部分；若两者不符，就表示出现散列冲突。

对出现散列冲突的处理，可以有二种办法。若 Key_1 、 Key_2 散列到同一个地址 A ，一种方法是把具有 Key_1 键符的内容放在 A 地址单元，而把具有 Key_2 键符的内容放在相邻地址 $A+1$ 单元内，这在硬件上便于实现，也便于查找。另一种是将具有 Key_2 键符的单元放在

$A_2 = H(H(\text{Key}_2))$ 地址单元内, 即再经一次散列。

对数据结构来说, 上述这些散列变换和Key的比较都是用软件来实现的, 但当把这种散列概念用到存贮体系的地址变换中来时, 为了满足速度的要求, 就得用硬件实现。上述散列变换是不难用硬件实现的, 例如折迭法中的按位加法就很容易实现 (见图5.43)。至于相乘法、余数法, 在原理上都可以经ROM或PLA实现, 随着VLSI的发展及硬件价格的下降, 散列变换的硬化就变得更加容易了。

除了 $A = H(\text{Key})$ 的用法外, 还有用散列法实现地址压缩, 即 $A' = H(A)$, 就是将长地址经散列“压缩”成窄地址, 这在概念上是一样的, 这种应用在 § 5 中再讲。

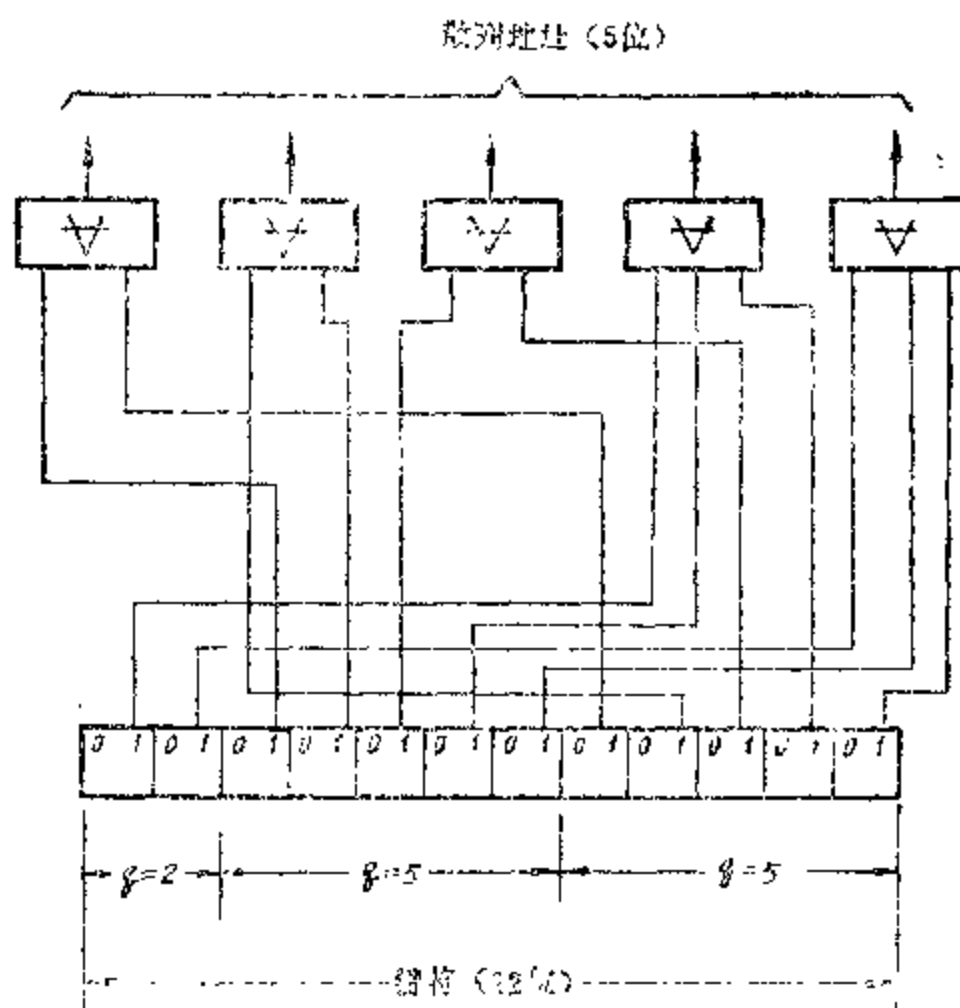


图 5.43 按位加折迭散列的硬件实现
(把12位键符变换成5位地址)

§ 4 替换算法及其实现

在“主存—辅存”存贮体系中, 主存的容量一般是比较辅存的要小得多, 因此必然会出现处理机要用到的指令或数据不在主存中的情况, 这时就产生页面失效 (故障), 要求从辅存中调进包括有这条指令或数据的页面。而且, 也必然会出现主存中所有页面已经全部被占用, 而又出现页面失效的情况, 这时将强迫腾出主存中的某个页面以接纳来自辅存要调的那页。那么, 按照什么算法 (规则) 来替换主存中的哪一页呢? 这就是所谓替换算法问题。

替换算法的确定要看它是否有利于提高存贮体系的性能。其中主要的是看命中率如何, 即要被访问的页已在主存的概率如何。显然, 二次页面失效之间的时间间隔愈大, 意味着命中率愈高。其次, 是看这种替换算法是否易于实现。

存贮体系中, 很多地方都要用到替换算法。除了在“主—辅”层次中用于主存页面的替换外, 还在“Cache—主”层次中用于Cache页面的替换以及地址变换的“快表—慢表”层次 (§ 5) 中用于快表内容的替换。下面虽然主要按“主—辅”层次中的应用来讲, 但它的原理同样适用于“Cache—主”层次和“快表—慢表”层次。只是其实现方法不同, 可能是全软的、全硬的或是软硬结合的。从六十年代存贮体系出现后, 对替换算法的研究就成了重要课题。

4.1 替换算法的分析

4.1-1 几种替换算法

一种简单、容易的算法是随机地选择被替换的页, 称为随机算法 (Random, RAND)。

它是用随机数产生器（软的或硬的）来决定被替换的页号。如果页面的使用是随机性的，那随机策略当然是可用的。但正如前面 §2 关于程序的局部性已经分析过的，对程序的访问不是随机的，而是簇聚的；由于随机法并没有利用到主存使用的“历史”信息，当然反映不了这种簇聚性，即前述工作区的特性。决定哪一页被替换与主存内各页原使用状况无关，就很可能把紧接着就要用的页面替换出去，把一段时期用不着的却保留下来，从而会带来很低的命中率。那么能反映过去使用状况的算法应是怎么样的呢？

一种算法是采用把驻留在主存中时间最长的那一页替换出去的先进先出（First-In First-Out, FIFO）策略。其出发点是最早进入的页很可能今后不再需要了。这种算法容易实现，例如，可给各页配一个计数器（软的或硬的），每当主存装入或替换时，除装入或替换的该页外，所有各页的计数器值都加“1”，通过检测各页的计数器值，操作系统按哪个计数器值最大判定驻留最长的是哪一页。先进先出算法虽然利用了有关主存使用的历史情况，但并没有能完全正确地反映程序的局部性。例如，最初进来的页若包含着循环，那怕此循环是经常用到的，但由于是最早进来就会被不合理地替换出去了，以至很快要用到时，产生页面失效而降低了系统效率。那么，如何能使替换算法更好地反映程序的局部性呢？如果能把近期最少使用的页替换出去就能比较正确地反映程序的局部性特点，这就是近期最少使用（Least Recently Used, LRU）算法。但由于其所需的信息状态较多，实现较困难，一般都采用它的变形，这就是我们在 §2.1 关于程序的局部性和工作区所举的例子中，讲过的“最久没被访问过”页面的概念。因此，一般的 LRU 算法实际上就是替换近期内最久没被访问过的页（即主存中各页，在近期内谁最久没被访问过就替换谁）。它是当前普遍采用的替换算法，实现起来比 FIFO 复杂一些，因为需要保存主存各页的使用状况。

然而，这种“近期”毕竟是指的过去了的近期，是根据过去的近期使用状况预估未来近期中哪一页可能不被使用，来把它替换出去。那么，能否真正根据未来的使用状况，把未来近期不会用到的页面替换出去呢？例如主存有 A、B、C、D 四个页面，在时间 t_i 看，如果在 t_i 之后，A 在 t_j 时用到，B 在 t_k 时用到，C 在 t_l 时用到，D 在 t_m 时用到。那么，当需要把新的一页 E 调入时，就应把 $t_j - t_i$ 、 $t_k - t_i$ 、 $t_l - t_i$ 、 $t_m - t_i$ 中值最大的那个页面替换掉。但是我们怎么能在 t_i 时预知到 t_j 、 t_k 、 t_l 、 t_m 时各页面的使用状况呢？除非是把程序在机器中运行二次，第一次是分析地址流，第二次才是真正运行，而这是不现实的。既费时又费事，在编译和执行过程中是几乎不可能取得为实现这种算法所需要的信息。因此，只是一种理想的算法，称为优化替换算法（Optimal Replacement Algorithm, OPT），只能作为衡量其它各种算法优劣的标准。

有时，在某些替换算法的基础上，增加判别该页是否修改过（被写过）的功能。如果没有写入过，由于辅存中原来有其付本，被替换时就可不必先写回辅存，而一旦改写过，被替换时必须将它先写回辅存，这需要化费相当的时间。因此，替换页的选择还可加上“未访问过且未修改过”、“已访问过但未修改过”、“已访问过且已修改过”等条件。

下面对 FIFO、LRU 和 OPT 进行比较。为了评价各种替换算法的优劣，多用模拟方法，即对典型的页地址流，分析各种算法的命中率高低来评价其好坏。当然，除了替换算法之外，影响命中率的_{因素}还有地址流的状况、页面的大小和主存的容量等。

设有一个程序 Q，总共有 5 页，分别为 $P_1 \sim P_5$ ，其页地址流为：

时间 t	t ₁	t ₂	t ₃	t ₄	t ₅	t ₆	t ₇	t ₈	t ₉	t ₁₀	t ₁₁	t ₁₂
使与页面 P _i	P ₂	P ₃	P ₂	P ₁	P ₅	P ₂	P ₄	P ₅	P ₃	P ₂	P ₅	P ₂

假设分配给该道程序的实存有三页。图 5.44 表示 FIFO、LRU、OPT 替换策略对这三页的使用情况，其中按所用算法选为替换页的用星标（*）标记。可以看到，P₂、P₅ 页是最经常要用到的，LRU 法能做到对这二页的命中率和 OPT 法的（在实存为三页时所可能达到的最大值）一样，但 FIFO 法要差。对此例的页地址流，FIFO 的页命中率最低，LRU 的页命中率非常接近于 OPT 的，在实际的程序运行中一般也是这样的。

然而，对某种页地址流，LRU 法也可能和 FIFO 法的一样糟。举例来说，如果一个循环程序所需页数大于分配给它的实存页数，就会发生这种情况。如图 5.45 所示，若循环程序需 P₁~P₄ 四页，而实存只有三页，则 LRU 法与 FIFO 法的命中率均是零，而 OPT 法命中三次。因此，无论是 FIFO，还是 LRU，对此例都是频繁、连续地产生页面失效，使

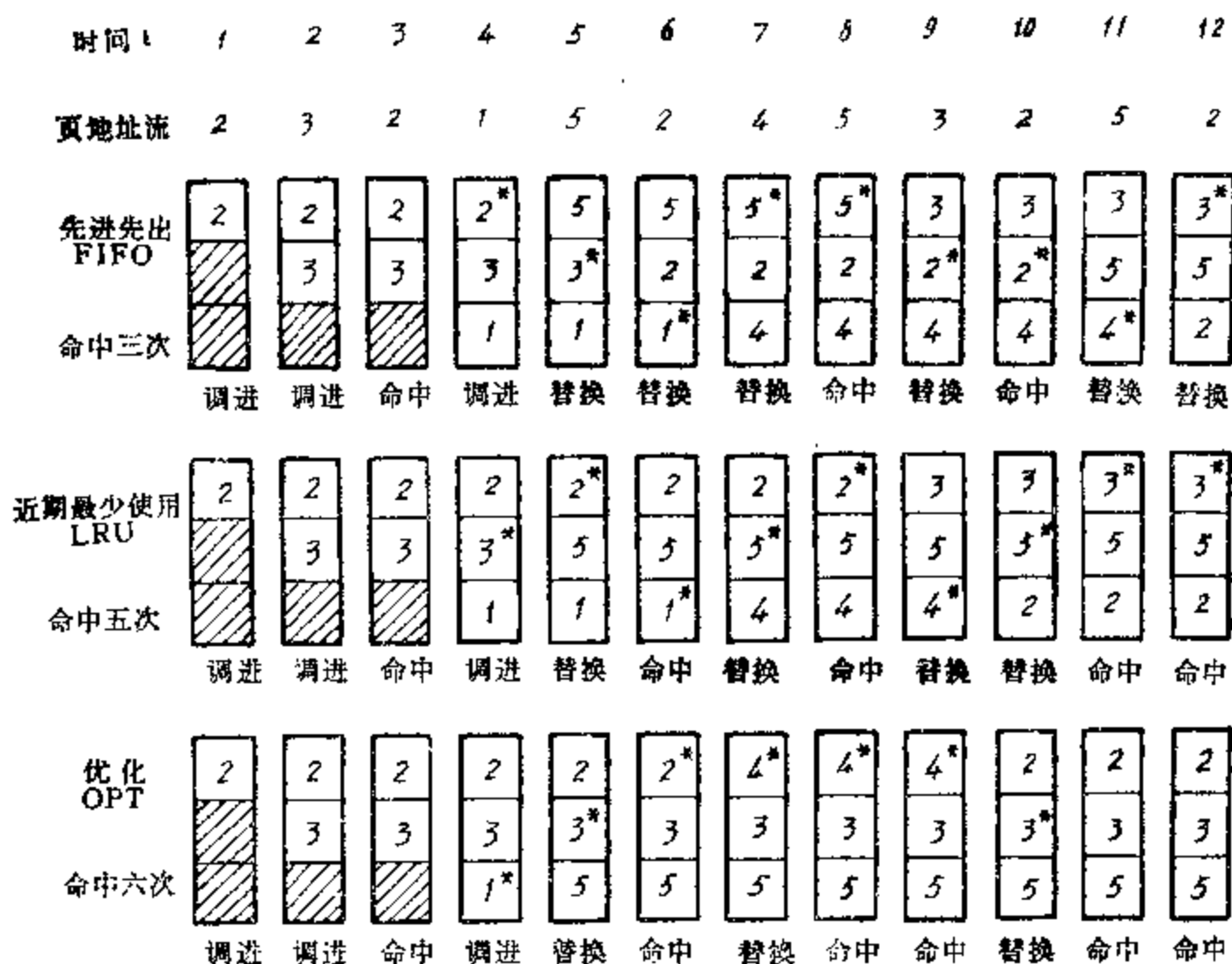


图 5.44 三种替换算法对同一页地址流的替换状况

系统效率显著下降。这种连续出现页面失效，使得几乎没有有效运算时间的现象称为颠簸，这当然是不希望有的，它是评价存贮管理和存贮体系性能好坏的一个重要标志。可以看出，如果分配给此程序的实页数增加一页，则可使命中率显著增大。后面还要证明，对 LRU 算法，实页数的增加必然会使命中率增大，而 FIFO 算法却不一定。

这样，对多道程序，如果分配给各个用户的实存页数是可以调节的，使得在某道程序的页面失效次数达到某个范围时，能动态、自适应地调节分配给该道程序（用户）的实页数，以使其命中率得到提高。如上例，当失效次数达到某个范围时，若自动使分配给该程序的实

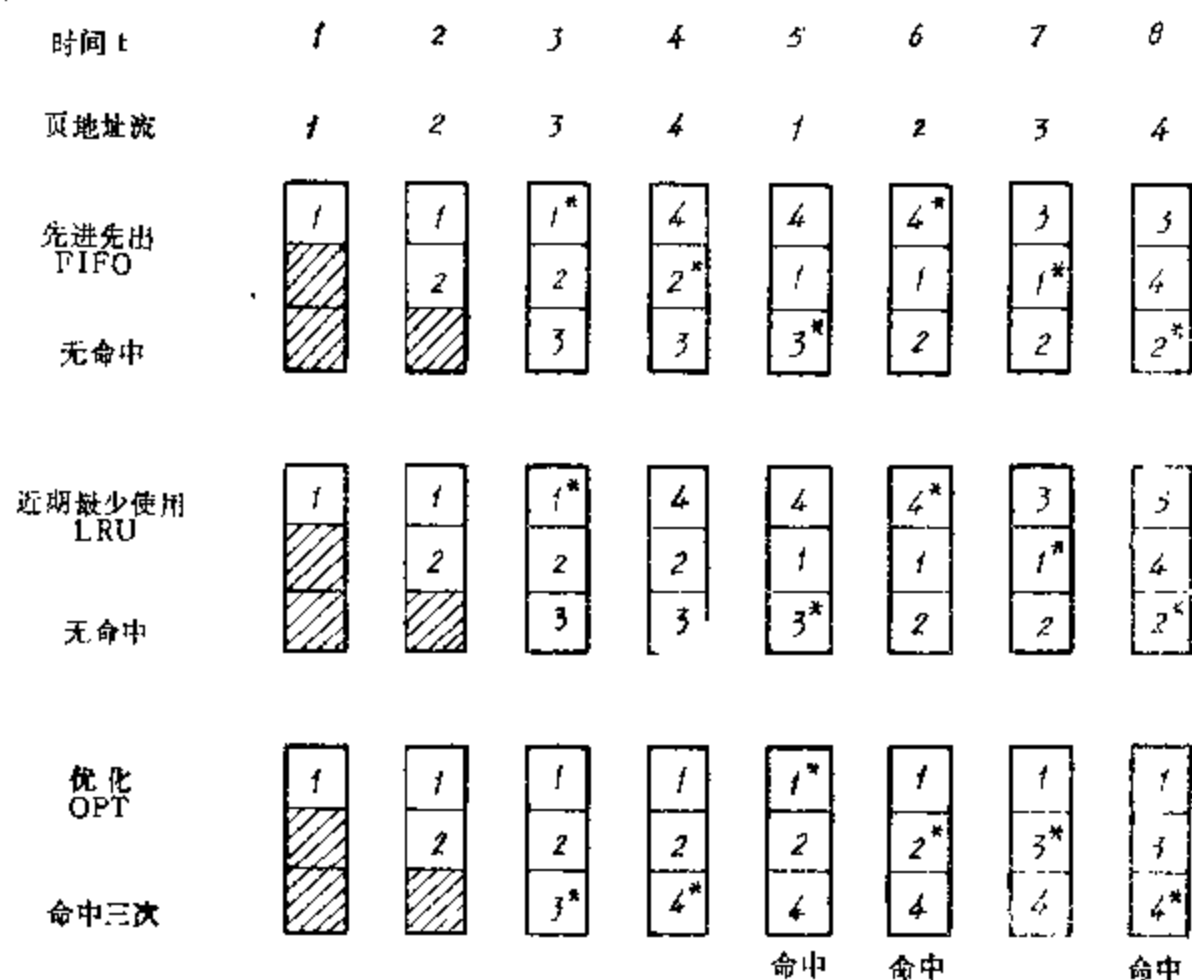


图 5.45 对这种页地址流 LRU 与 FIFO 一样糟

页数增加一页，就会使之后的命中率显著提高。这种算法称为页面失效频率 (Page Fault Frequency, PFF) 法。

4.1-2 堆栈型替换算法的特点

前面已经说过，影响命中率的因素有多种，要对所有这些都进行模拟，工作量是非常大的。为此提出了若干能优化存贮体系设计、减少模拟工作量的分析模型，其中堆栈处理技术可用于分页系统的分析，适用于采用堆栈型替换算法的系统。

堆栈型替换算法的定义如下：

设 A 是长度为 L 的任何页地址流， t 为已处理过 $(t-1)$ 个页面的时间点， n 为分配给该地址流的实存页面数， $B_t(n)$ 表示在 t 时间点，在 n 页的实存中的页面集， L_t 表示到 t 时已遇到过的地址流中相异页的页数，如果替换算法具有下列包含性质，即：

$$\begin{aligned} \text{当 } n < L_t \text{ 时, } B_t(n) &\subset B_t(n+1) \\ n \geq L_t \text{ 时, } B_t(n) &= B_t(n+1) \end{aligned}$$

则此替换算法为堆栈型的。

LRU 型算法，由于它在实存中保留的是 n 个最近使用过的页面，它们又总是包含在 $n+1$ 个最近使用过的页面内，所以 LRU 是堆栈型算法。同样，OPT 也是堆栈型算法。然而，FIFO 不是堆栈型算法。例如，对下述页地址流

$$A = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$$

图 5.46 表示实页数为 3 和 4 时，采用 FIFO 法进行处理的情况。看出，在各个时间点，它不能全部满足上述的包含性质。例如， $B_7(3) = \{1, 2, 5\}$ ，而 $B_7(4) = \{2, 3, 4, 5\}$ ，所以， $B_7(3) \not\supset B_7(4)$ 。因此 FIFO 不是堆栈型算法。而且由此例看出，FIFO 法随着 n 值的

增大，命中率不仅没有提高，反而下降了。

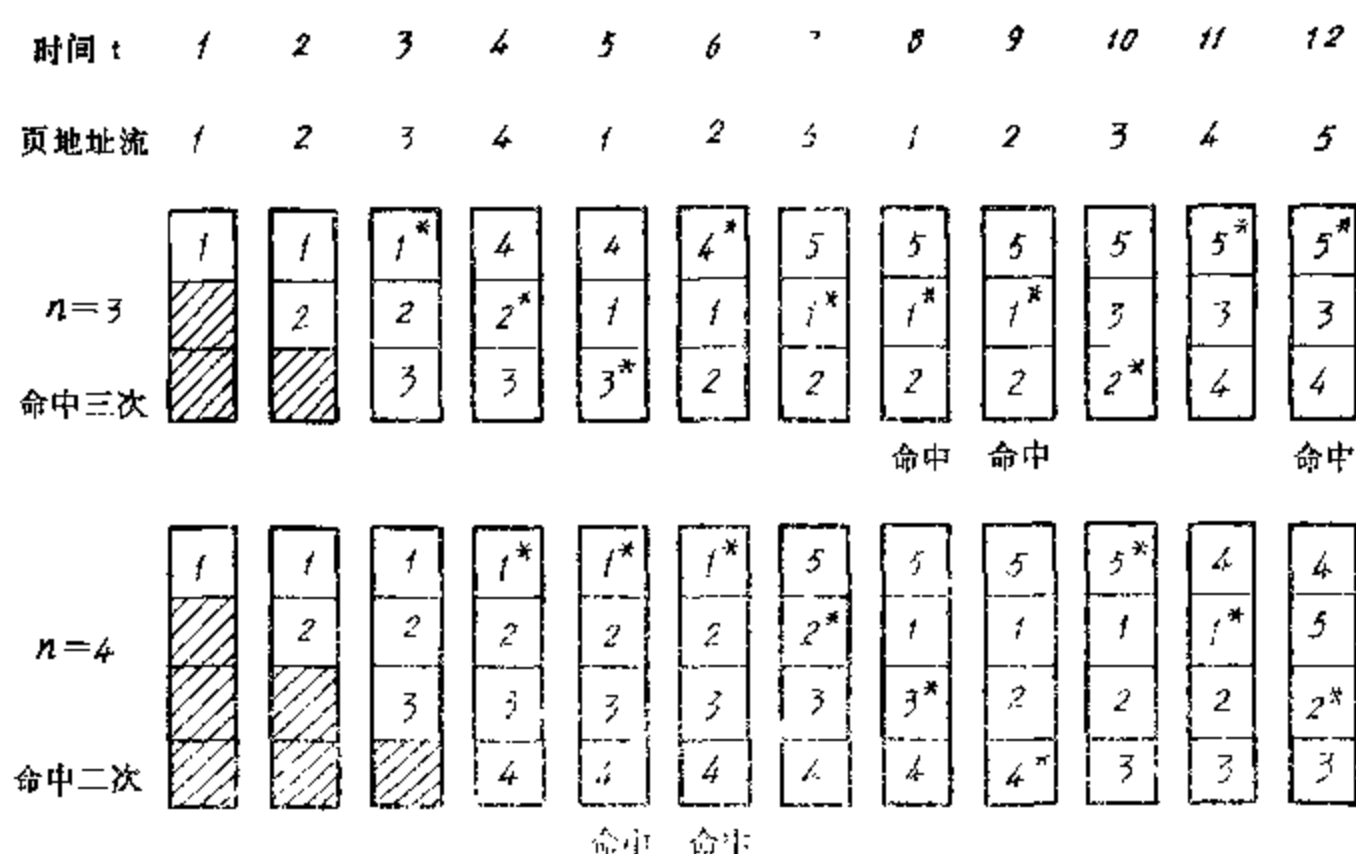


图 5.46 对二种实页数 n，FIFO 法的替换状况

对于堆栈型算法，只需采用堆栈处理技术对地址流模拟处理过一次，就能获得对此地址流在不同实存容量时的命中率。从而，能大大节省存贮体系的设计工作量。研究分析模型的目的就在于此。

用堆栈处理技术对地址流进行模拟处理时，主存在 t 时的状况用堆栈 S_t 表示， S_t 是 $S_t(1), S_t(2), \dots, S_t(L_t)$

的有序集（前已说过， L_t 是到 t 时地址流中相异页的页数）， $S_t(1)$ 是栈顶，第二项是 $S_t(2)$ ，第三项是 $S_t(3)$ ，……。

按前述堆栈型算法的定义，必然是：

对 $n < L_t$ ， $B_t(n) = \{S_t(1), S_t(2), \dots, S_t(n)\}$

对 $n \geq L_t$ ， $B_t(n) = \{S_t(1), S_t(2), \dots, S_t(L_t)\}$

这样，对容量为 n 的主存，它的状态就是由 S_t 的顶 n 项表示。因此，对页地址流 A 在 t 时的 A_t 页是否命中，只需看 S_{t-1} 的顶 n 项中是否有和 A_t 相符的，若有则是命中。所以，经过一次模拟处理获得 $S_t(1), S_t(2), \dots, S_t(n)$ 之后，就能知道对应所有不同 n 值的命中率。从而能很快地为主存容量的确定提供依据。

对不同的堆栈型替换算法， S_t 各项的改变过程是不同的，下面讲 LRU 法的。它把主存中刚访问过的页号置于栈顶，而把最久没有访问过的页号置于栈底。确切地说，对于 A_t ，若 $A_t \notin S_{t-1}$ ，则把 A_t 压入堆栈使之成为 $S_t(1)$ ，而 $S_{t-1}(1)$ 成为 $S_t(2)$ ， $S_{t-1}(2)$ 成为 $S_t(3)$ ，……，即各项都下推一个位置；若 $A_t \in S_{t-1}$ ，则把它由 S_{t-1} 中取出，压入栈顶，成为 $S_t(1)$ ，在 A_t 之下的各项位置不动，而之上的各项都下推一个位置。图 5.47 是对应于图 5.44 页地址流的 S_t 。

由图 5.46 的 S_t 可确定对应这个地址流，当主存容量 n 取不同值时的命中率。对不同的 n 值，

若 $A_t \in S_{t-1}$, 则是命中;

若 $A_t \notin S_{t-1}$, 则是不命中。

例如, 对 $n=4$, 其 $S_6 = \{5, 1, 2, 3\}$, 因为 $A_6 = 2 \in S_6$, 所以命中; 但对 $n=2$, 其 $S_6 = \{5, 1\}$, 因为 $A_6 = 2 \notin S_6$, 所以不命中。这样, 就可算出各个 n 值的命中率 H^* , 如下表所示:

时间 t	1	2	3	4	5	6	7	8	9	10	11	12
页地址流 A	2	3	2	1	5	2	4	5	3	2	5	2
$S_t(1)$	2	3	2	1	5	2	4	5	3	2	5	2
$S_t(2)$		2	3	2	1	5	2	4	5	3	2	5
$S_t(3)$				3	2	1	5	2	4	5	3	3
$S_t(4)$					3	3	1	1	2	4	4	4
$S_t(5)$							3	3	1	1	1	1
$S_t(6)$												
$n=1$												
$n=2$			命中									命中
$n=3$			命中			命中		命中			命中	命中
$n=4$			命中			命中		命中		命中	命中	命中
$n=5$			命中			命中		命中	命中	命中	命中	命中

图 5.47 使用 LRU 法对页面地址流进行堆栈处理

n	1	2	3	4	5	>5
H^*	0.00	0.17	0.42	0.50	0.58	0.58

可见 LRU 法的命中率随分配给该道程序的实页数的增加而单调上升。其它堆栈型算法也是如此。但对 FIFO 这种非堆栈型算法, 实页数的增大有时反倒可能降低命中率。

4.2 替换算法的实现

前面已经讲了 FIFO 算法的实现方法, 因此本节只讲述 LRU 替换算法的实现。

如何找出近期中最久没有被访问过的页, 可以有多种方法。下面介绍使用位法、堆栈法和比较对法。

4.2-1 使用位法

这种方法主要在“主—辅”层次的虚拟存储器中使用。

给每个实存页面配一个使用位(访问位), 开始时所有页的使用位全为零, 只要某个实页的任一单元被访问过, 就由硬件自动地置该页的使用位为 1。这样, 当需要替换时, 只需

替换使用位为零的页即可。这在原理上是简单的，但实现起来还会有很多具体问题。

前面讲过的页表是对应每个程序的，而使用位是对应实存页的，显然不能把它放在页表内，而应放在对应主存所有实页的表内。操作系统为主存管理设置了“主存（实存）页面表”，记录实存中每页的状况，使用位就装在这个表内，表内其它项的用途以后再讲，它的构成如图 5.48 所示。IBM370 设置了能改变表中每个实页各种状态的特权指令。

实 存 页 面 表						
实页号	占用位	程序号	段页号	使用位	程序优先位	H
0 #						
1 #						
⋮						
⋮						
15 #						
⋮						

图 5.48 实存页面表

实存页面表存于主存。由于实页号是顺序的，因此该项可以略去，而用相对于表起点的相对位置来表示实页号。为了能由实存页面表判定哪些实页已被占用，哪些实页是空的，设立“占用位”。占用位为“1”表示该实页已被占用，占用位为“0”表示该实页空着。至于由哪个程序的哪个段、哪个页占用则由程序号和段页号表示。

对于全相联映象，调入页可进入对应于实存页面表中任何占用位为“0”的实页，一旦装入实存某页后，该实页的占用位被置成“1”。其实，对于 IBM370，把某个虚页调入实存是通过“页面联接”指令，把由该指令地址码指明的虚页按替换算法的规定与某个实页“相联”，并把该虚页的程序号和段、页号置入实存页表中对应该实页的行。当所有实页的占用位都是“1”，而又发生页面失效时，就产生页面替换问题。这时就要用到使用位，替换使用位为“0”的页面。可是，如果是在全部页都使用过，即使用位全部为“1”的情况，再发生页面失效，那就无法判定究竟哪页该被替换。所以，使用位为全“1”的情况是不许出现的。为此，可以采用的办法之一是一旦使用位进入全“1”时，就随即由硬件强制全部使用位为“0”。从概念上来看，就是说近期最少使用的“期”是从上一次使用位为全“0”起到这一次使用位为全“0”的这段时间。显然这个“期”的时间长短是随机的，故称为随机期的办法。

与随机期法相对的是定期法，它定期地使全部使用位为0。为此，可给每个实页配一个“没使用过计数器”H（或称历史位），定期地每隔 Δt 时间（例如每隔几毫秒、几秒或几分）扫视所有使用位，进行记录。例如，可以对使用位为0的页，加“1”H，并让使用位仍保持为0；而对使用位为1的页，置“0”H，同时，置“0”使用位。这样，扫视结束时所有使用位都成了“0”，开始一个新的 Δt 期，但原有使用位的状态都已被各自的H记录下来。因此，H值愈大的，表明它是最长时期未被使用过的，就应是可被替换的候选页。实质上，使用位是反映一个 Δt 期内的使用情况，H是反映多个 Δt 期内的页面使用情况。

上面分析了随机期和定期的二种方法，它们各有其优缺点。由于页面失效时主存与辅存间的页传送时间本来就很长，因此，如何判使用位和H的状态，如何得知使用位已进入全“1”以及如何置全“0”等一般是软硬结合地实现。

此外，还需要注意，有些程序（如操作系统中的常驻部分）是要常驻主存根本不允许被替换的。为此，可以在实存页面表中设置“程序优先使用位”，使得优先级高的（优先级可以有几级，用多个“优先使用位”表示）某些实页，它们的使用位不准被置“0”，这些实页就不会被替换。还有一些情况，在图5.48的主存页面表中不能很好反映。例如，上面讲的是只要某页中一个单元被访问过一次该页的使用位就被置“1”，这显然是粗糙的。因为，对于一页中只有一个单元只被访问过一次与一页中多个单元被访问过或被访问过多次比较起来，当然应选择前者作为替换页。但上述的使用位办法不能区分这二者。在§4.1讲过，替换页的选择还应考虑该页是否被修改过，因此在主存页面表还应设置“修改位”。

4.2-2 堆栈法

在§4.1-2中，讲过LRU法是堆栈型替换算法，也讲了对LRU法堆栈 S_i 的各项应如何改变，其结果是栈顶恒为近期最近访问过的页的页号，而栈底恒为近期最久没有访问过的页的页号，即替换页的页号。按此，可实际组成有 2^{n_v} 项（行）的这种堆栈，如图5.49所示，堆栈的容量等于实存页面数。按§4.1-2所讲，把被访问的实页地址与堆栈中各行的 n_v 值相联比较，如果没有相符的，则把此页地址压入堆栈，且把堆栈的所有项都顺次下推一个位置；如果有相符的，也把此页地址压入堆栈，这相当于把堆栈中存放的 n_{v_i} 上移到栈顶，且只是由存 n_{v_i} 的行到栈顶的那些项下推一个位置。这样，当堆栈全被装满（即实存全被占用）后，又出现页面失效时，栈底的 n_v 就是最久未被用过的可被替换的实存页号。显然，对于全相联映象，只需有一个这种堆栈；而对组相联映象，则每组都需要有一个容量较小的这种堆栈。因为这种堆栈要求有相联比较和既能全下推又能部分下推以及能由中间取掉一项等功能，如果用硬件实现，只能用于实现组相联的LRU替换。对于全相联的主—辅层次，

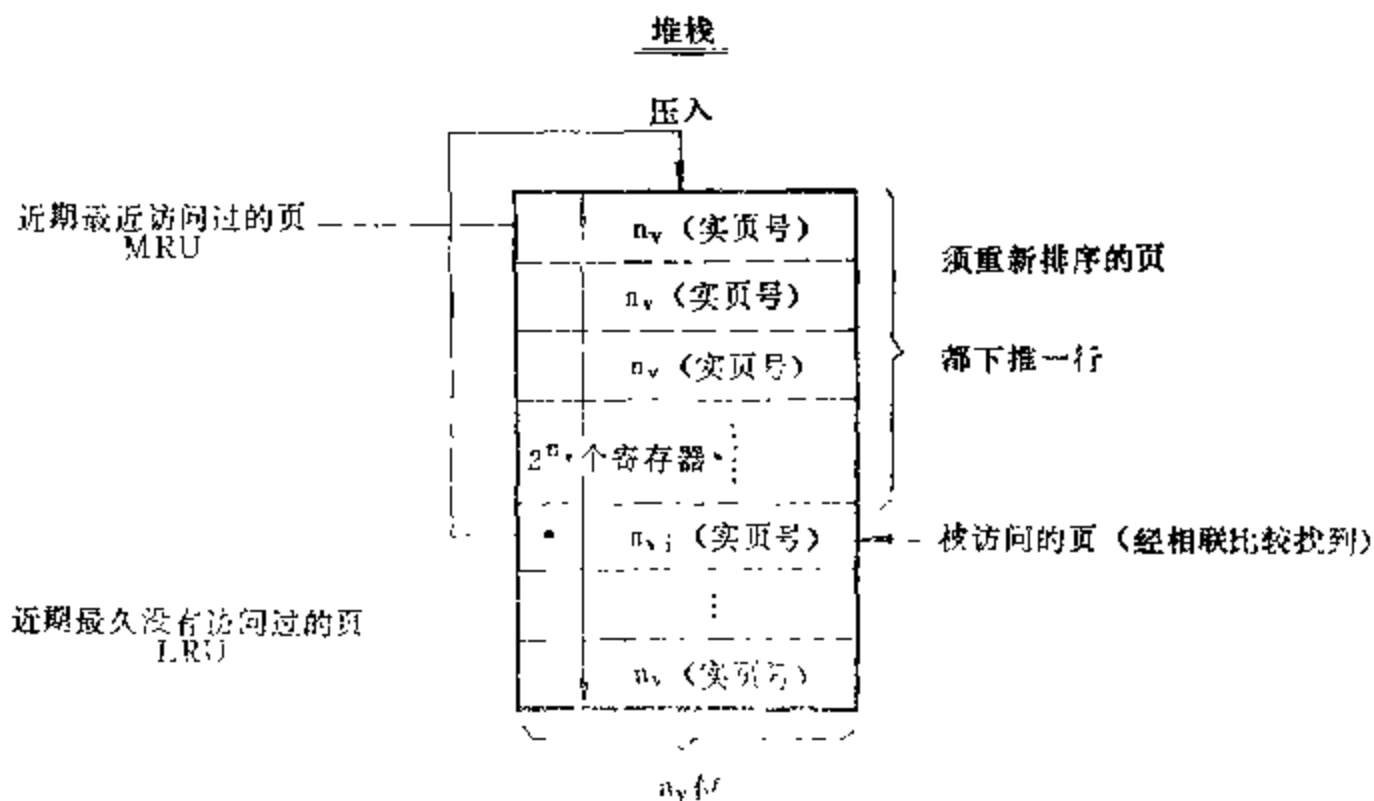


图 5.49 LRU 法经堆栈实现（需有相联比较功能）

如果采用这种替换用堆栈可用软件实现。

这种堆栈法，各实页被访问过的先后次序是反映于几何位置，即栈顶恒是最近刚访问过的实页号，下一行被访问过的次序比它早，如此直至栈底是最久没有访问过的。与它相对的办法是用几何位置反映实页号，而访问次序用数值表示，这样就需用如图 5.50 那样的寄存器组，各个实页的访问次序存在对应的寄存器内，寄存器组需有 2^{n_v} 个寄存器，每个寄存器为 n_v 位，才能表示从 1 到 2^{n_v} 的次序，愈是最近访问的，其次序值愈小。替换页的确定也

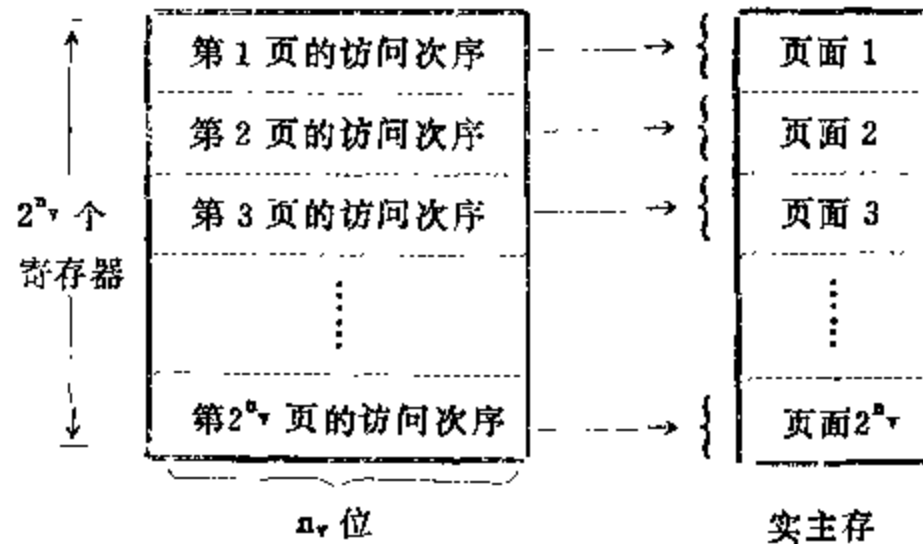


图 5.50 LRU 法经寄存器实现 (需有相联比较功能)

需用相联比较，但不是相符比较，而是找出最大值所在的寄存器，由它的几何位置求得替换页号。

4.2-3 比较对法

上述堆栈法，需要有相联比较功能，具有相联比较功能的硬件其速度较低，也比较贵，那么，能否不用相联比较，而用一般的门、触发器来实现 LRU 替换算法呢？比较对法就是其中的一种。

比较对法的基本思路是让各个页成对组合，用触发器状态表示每个比较对内的访问次序，再经与门求得 LRU 页。

例如有 A、B、C 三页，互相之间可组合成 AB、BA、AC、CA、BC、CB 六对，其中 AB 和 BA，AC 和 CA，BC 和 CB 是重复的，所以只需取 AB、AC、BC 三对，各对内的访问顺序分别用“对触发器” T_{AB} 、 T_{AC} 、 T_{BC} 表示。 T_{AB} 为“1”，表示 A 比较 B 更近被用过； T_{AB} 为“0”表示 B 比 A 更近被用过。 T_{AC} 、 T_{BC} 也类似定义。在这三对之间，如果 $T_{AB}=1$ ， $T_{AC}=1$ ， $T_{BC}=1$ ，则为 A 比 B 近，A 比 C 近，B 比 C 近，访问次序为 ABC，即最近使用过的为 A，最久使用过的为 C，如果 $T_{AB}=0$ ， $T_{AC}=1$ ， $T_{BC}=1$ ，则其使用次序为 BAC，C 为最久使用过的页。

上述关系用布尔代数式表示为：

$$C_{LRU} = T_{AB} \cdot T_{AC} \cdot T_{BC} + \overline{T_{AB}} \cdot T_{AC} \cdot T_{BC} = T_{AC} \cdot T_{BC}$$

同理可得，

$$B_{LRU} = T_{AB} \cdot \overline{T_{BC}}$$

$$A_{LRU} = \overline{T_{AB}} \cdot \overline{T_{AC}}$$

显然，这些式子可用与门实现，如图 5.51 所示。

下面分析比较对法所用的硬件。看出：每页需有一个与门；每个与门接收与它有关的触发器来的输入，例如 A_{LRU} 与门要有从 T_{AB} 、 T_{AC} 来的输入， B_{LRU} 要有从 T_{AB} 、 T_{BC} 来的输入，而与每页有关的对数为页数减去 1，所以与门的扇入数是页数减去 1。若 P 为页数，则可能的比较对数为 $P(P-1)$ ，但这些对中，有一半是重复的，所以，所需触发器数即比较对的对数为 $\frac{P(P-1)}{2}$ 。

页数与对数、门数、门输入端数的数值关系如下表：

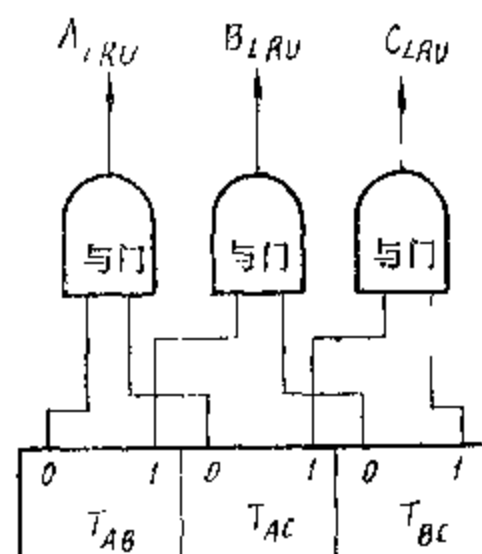


图 5.51 用比较对法实现 LRU 算法

页 数	3	4	8	16	64	256	...	P
比较对数 (触发器数)	3	6	28	120	2016	32640	...	$\frac{P(P-1)}{2}$
门 数	3	4	8	16	64	256	...	P
门输入端数	2	3	7	15	63	255	...	$P-1$

因为触发器数随页数如此迅速地增加，所以比较对法只适用于采用组相联映象的主—Cache 层次。在页数少时它比前述堆栈法或使用位法易于实现，因为它不需要有相联比较功能，可用通常的门、触发器实现。

显然，在每次访问后，需改变比较对触发器的状态，其规则并不复杂。

拿上述 A、B、C 三页为例。每次访问 A 后需改变和 A 有关的比较对触发器状态，以反映 A 比 B 近，A 比 C 近，即需置 “1” T_{AB} 、 T_{AC} ；同理，访问 B 后需置 “0” T_{AB} ，置 “1” T_{BC} ；访问 C 后需置 “0” T_{AC} 、 T_{BC} 。由此可定出各个对触发器的输入控制逻辑。对于有更多的页数，可用同样概念推出。

堆栈法和比较对法是实际中采用的办法，IBM370/165、360/195 的 “Cache—主” 层次采用比较对法，我国的 905 机也是用此法；IBM370/168 的 “Cache—主” 层次所用办法类似于堆栈法。

综上所述，设计替换算法的实现应围绕：

- (1) 如何对每次访问进行记录，使用位法、堆栈法、比较对法所用记录方法都不同；
- (2) 如何根据所记录的信息来判定近期内谁最久没有被使用过。

还可看出，实现方法和所用的映象方法密切相关。例如，对于 “主—辅” 层次的全相联映象宜于采用使用位法或类似的方法，而不宜用堆栈法和比较对法；但对于 “Cache—主” 层次的组相联映象，因为组内页数较少，就宜用比较对法或堆栈法。替换算法的设计和其实实现密切相关，随着器件的改进尤其是相联存贮器的改进，已经而且必然会不断研究出新的更好的实现方法。

§ 5 虚拟存贮器

前面已经讲过,习惯上,虚拟存贮器指的是“主存——辅存”层次,它能使该层次具有辅存的容量,接近于主存的等效速度,接近于辅存的每位成本。它使得程序员可以按比主存大得多的空间来编制程序,即按虚存空间编址,只要主存容量大于某个最小值,那不论机器配备多大容量的主存,程序可不必作任何修改照样能运行。当然,主存实际容量的大小是会影响系统工作的效率,如果程序过大而主存容量过小,则解题速度会明显下降。

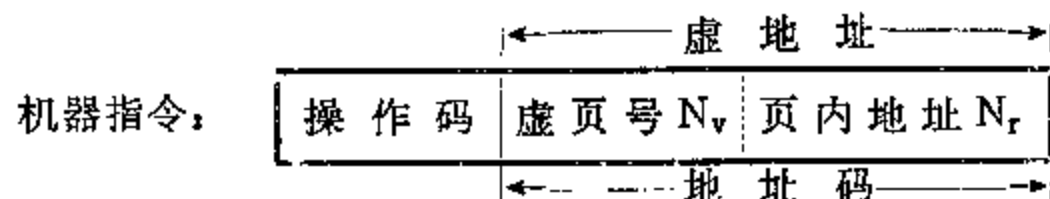
本节着重讲述虚拟存贮器原理和工作的全过程,并利用 Huffman 概念对各种情况、各个工作阶段的速度要求和实现方法进行分析。

5.1 虚拟存贮器原理

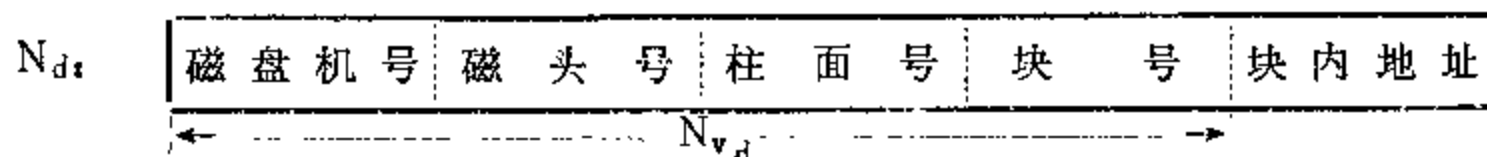
有关虚拟存贮器的基本概念,如虚地址到主存实地址的映象和变换、替换算法等,在前面已作了讲述,下面先讲述由虚地址到辅存实地址的变换以及多用户对虚拟存贮器的要求,然后再讲虚拟存贮器工作的全过程和实现中的某些关键问题。

5.1-1 虚地址到辅存实地址的变换

我们知道,对虚拟存贮器,程序员是按虚存空间编制程序,即按机器指令的地址码来编制,这个地址码就是虚地址,它由虚页号及页内地址组成,如下所示



这个虚地址实际上是辅存的逻辑地址,而不是辅存的实地址。以磁盘为例,按字编址的实地址 N_d 如下:

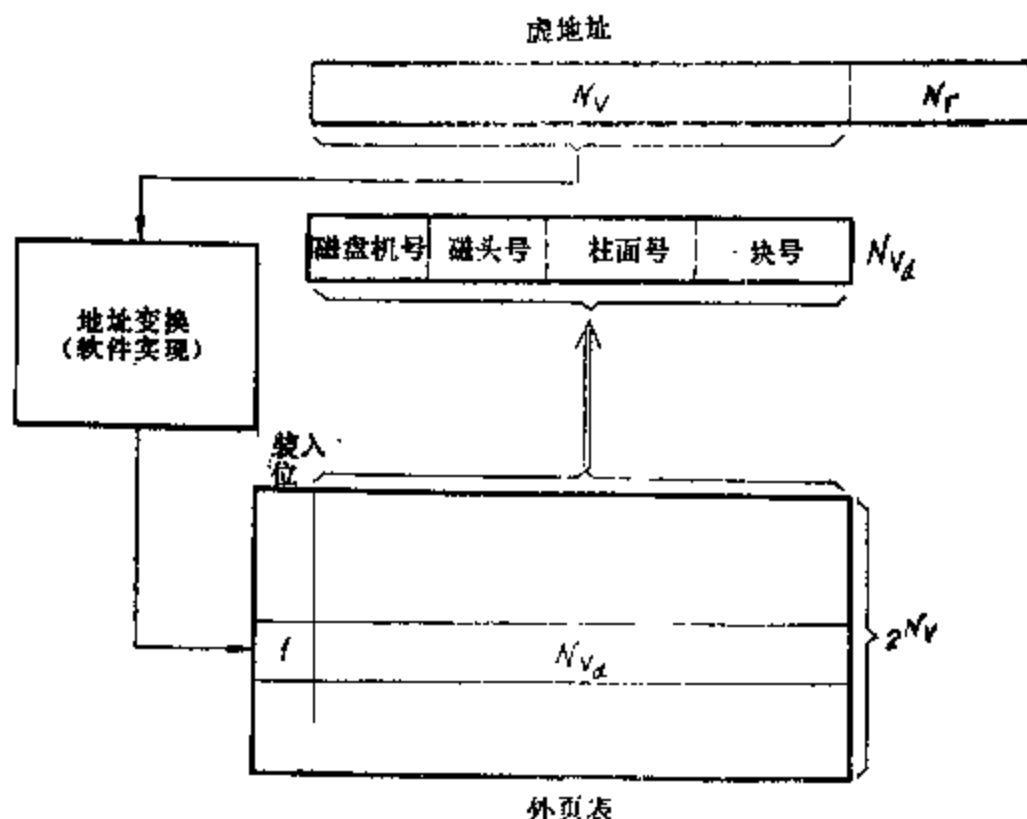


当然,辅存逻辑地址和辅存实地址二者不是一回事。因此在虚拟存贮器中还得有虚存空间到辅存实空间的地址变换。辅存一般是按信息块编址,而不是按字编址,若使一个块的大小等于一个虚页面的大小,这样就只需由虚页号 N_v 变换到 N_{vd} 即可完成虚地址到辅存实地址的变换。为此,可以采用类似前述页表的方式。我们把 N_v 变换到 N_{vd} 的表称为外页表,而把由 N_v 变换到主存实页号 n_v 的表称为内页表(即前述页表)。

显然外页表的容量也是 2^{N_r} ,也应采用前述表层次技术。前面讲过每访问一次主存都得进行虚地址到主存实地址的变换,但却只当出现页面失效时(其概率不到1%),才需调用外页表进行虚地址到辅存实地址的变换,以便由辅存将该虚页调入主存。因为访问辅存需经机械动作,所以速度低,本来就很费时间,因此对查外页表的速度要求也可较低,外页表可存在辅存,只在需查用时才调入主存,由虚地址到辅存实地址的变换是用软的办法,不必提

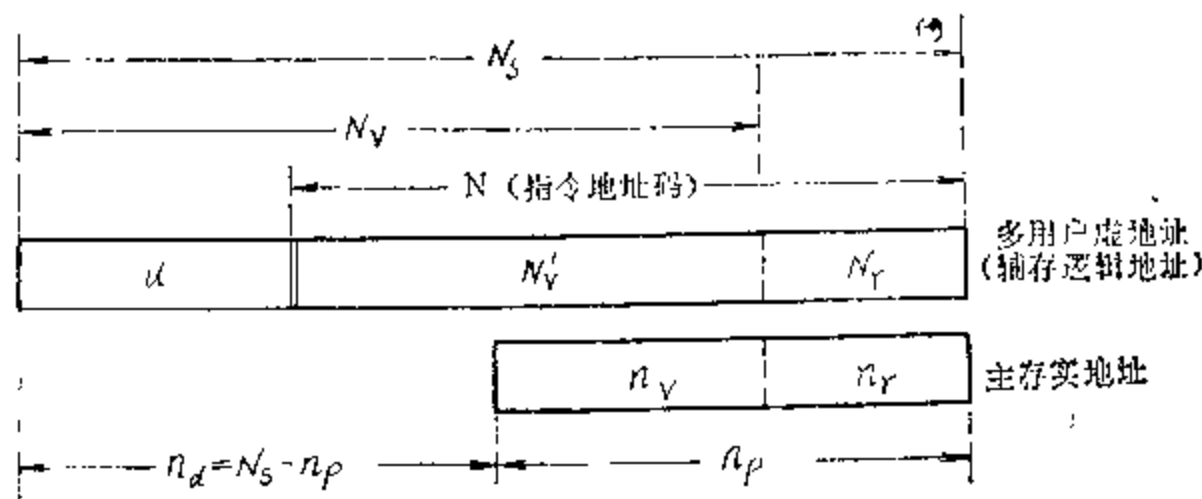
供专用的支持硬件。当然，当页面失效时，不是让处理机空等着该页由辅存调入主存，而是通过程序换道，切换到另一道已在主存的用户程序。虽然换道一般需要执行几百至几千条指令，但毕竟比起调页来说耗费的时间要小得多。

虚、辅地址的变换过程如图 5.52 所示。其中，装入位是表示该信息块是否已由海量存储器（如磁带）装入磁盘，当装入位为“1”时，外页表放的是该信息块（页面）在辅存（磁盘）中的实际位置，虚地址到辅存实地址的映象当然需是“全相联”的。



5.1-2 多用户虚拟存储器 图 5.52 用软件方法查外页表实现虚地址到辅存实地址的变换

上面我们是按多个用户共用一个总的虚存空间来讲的，虚存空间总共有 2^{N_V} 页，早期的虚拟存储器，多个用户（多道）只能分用这个虚存空间。这样，程序员不是具有与其它用户、其它道完全无关的编址空间，给程序的编制带来麻烦。所以，后来的虚拟存储器就设计成使每个用户都具有独立的有 2^{N_V} 个页面的虚存空间，而用另外的用户标志位 u （其宽度为 u 位）对应 2^u 个用户，来区分各个不同用户在辅存所占的空间。这样，辅存逻辑地址（即多用户虚地址） N_s 就由 u 和指令地址码 N 两部分组成如下：



多用户虚存空间能存得下 2^u 个 2^N 字的编址空间，多用户虚地址与主存实地址的差 $n_d = N_s - n_p$ ，现在的虚页号 $N_V = N_s - N_r$ ，而过去单用户中 $N_V = N - N_r$ ，它现在用 N_V' 表示。例如 IBM370，其 $N = 24$ 位， u 可达 24~32 位，因此 N 与 u 是分存于至少二个寄存器。

对于这种多用户的情况，由虚地址变换成主存实地址是将 $N_V = u + N_V'$ 变换成 n_V ，再拼接上 N_r 。

如果采用“全相联页式管理”查表法，每道程序（用户）有其内页表（ $2^{N_V'}$ 大），若该用户的内页表已在主存，则其地址变换可如图 5.53 那样处理。由 u 指明该用户内页表的起

点, 由 N_v' 指明某用户页表内的对应行, 查得实页号 n_v , 然后再与 N_r 拼接成主存地址。

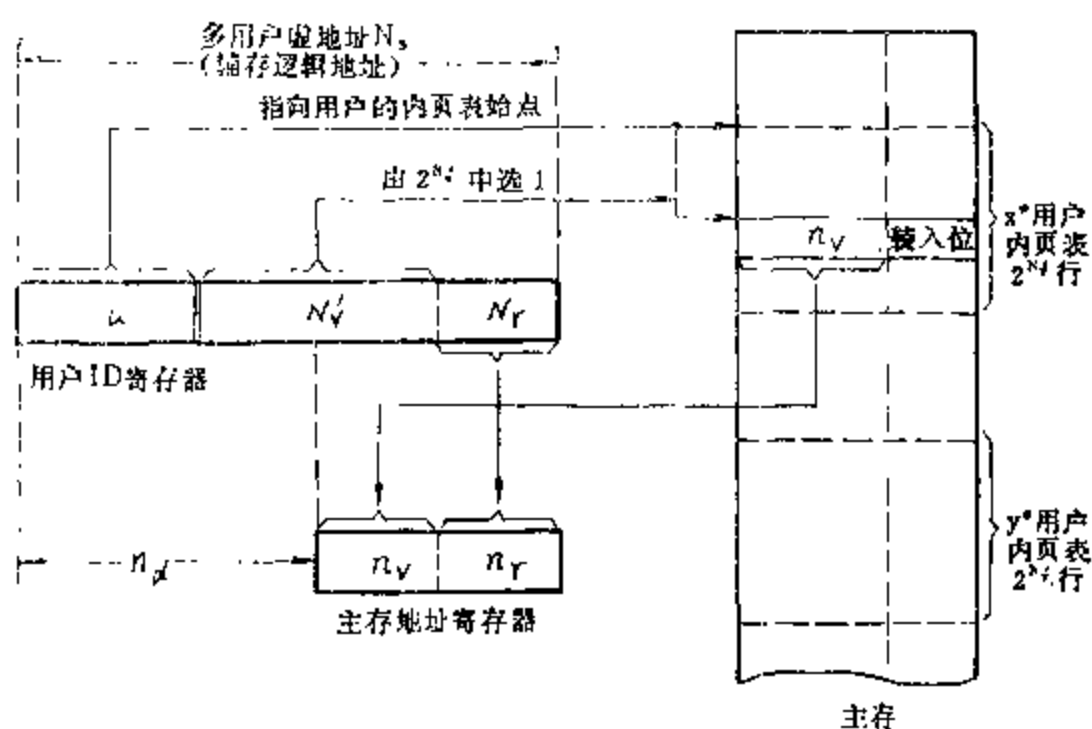


图 5.53 虚地址 N_v 变换成实主存地址, 每个用户一个内页表

对于“全相联段页式管理”, N_v' 需分成段号和页号, 若采用段表和页表的层次, 则可以把“u”作为各用户的段表始点, 如图 5.54 所示。

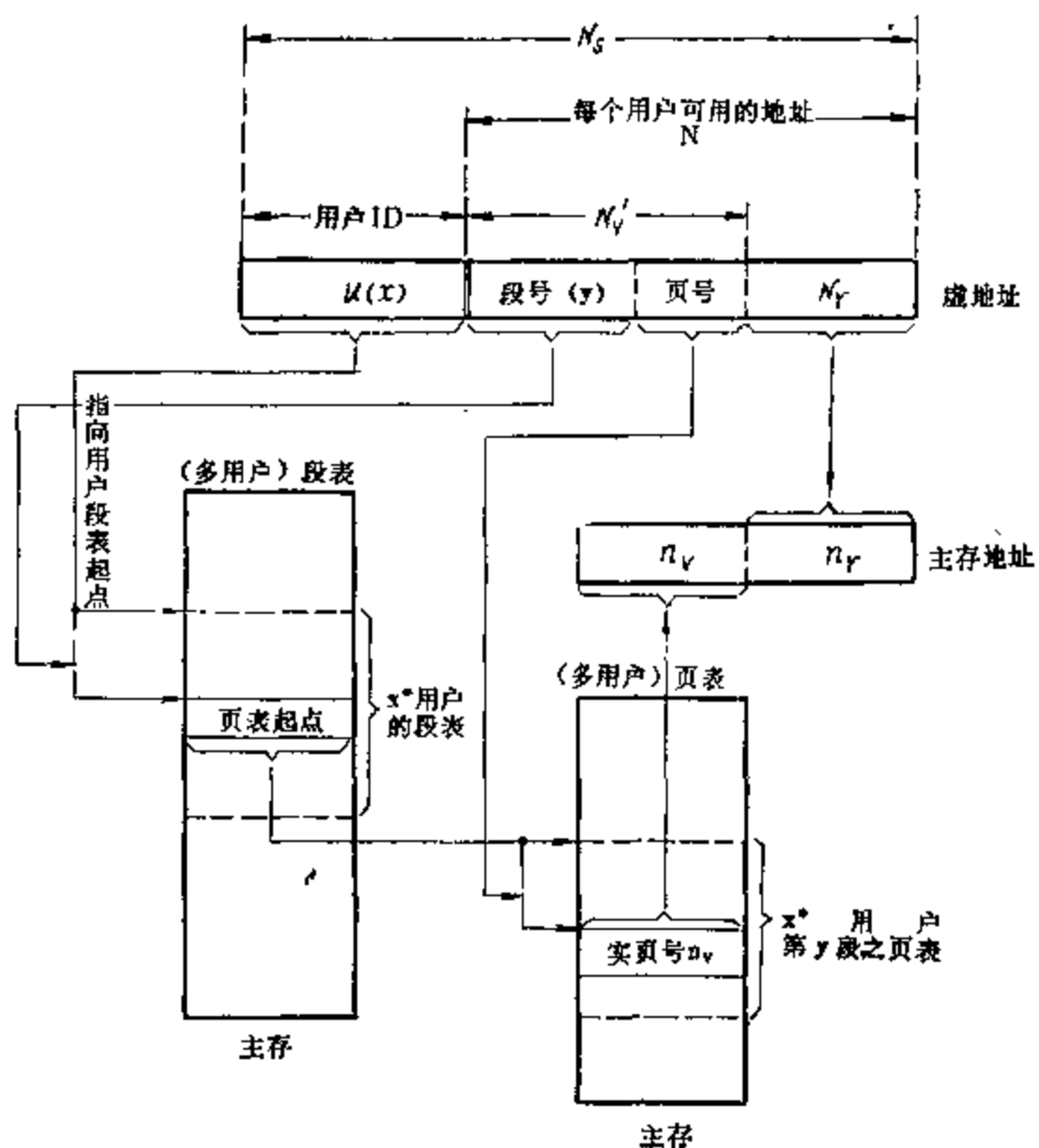


图 5.54 对多用户段页式管理, N_v 到主存地址的变换

注意, 对多用户虚拟存储器, 每个用户的编址空间和辅存空间(总的虚存空间)是不同的。

5.1-3 虚拟存贮器工作的全过程

现在将前述有关虚拟存贮器的各部分综合成如图 5.55。

在虚拟存贮器中每次访主存时，都需要将多用户虚地址变换成实主存地址①②，因此需要有虚页号变换成主存实页号的内部地址变换，例如通过查内页表。当对应该虚页的装入位为“1”时，就按主存实地址 n_p 访主存③；如果对应该虚页的装入位为“0”，表示该页未在主存中，就产生页面失效中断④，经它从辅存中调页。先通过外部地址变换⑤，例如查外

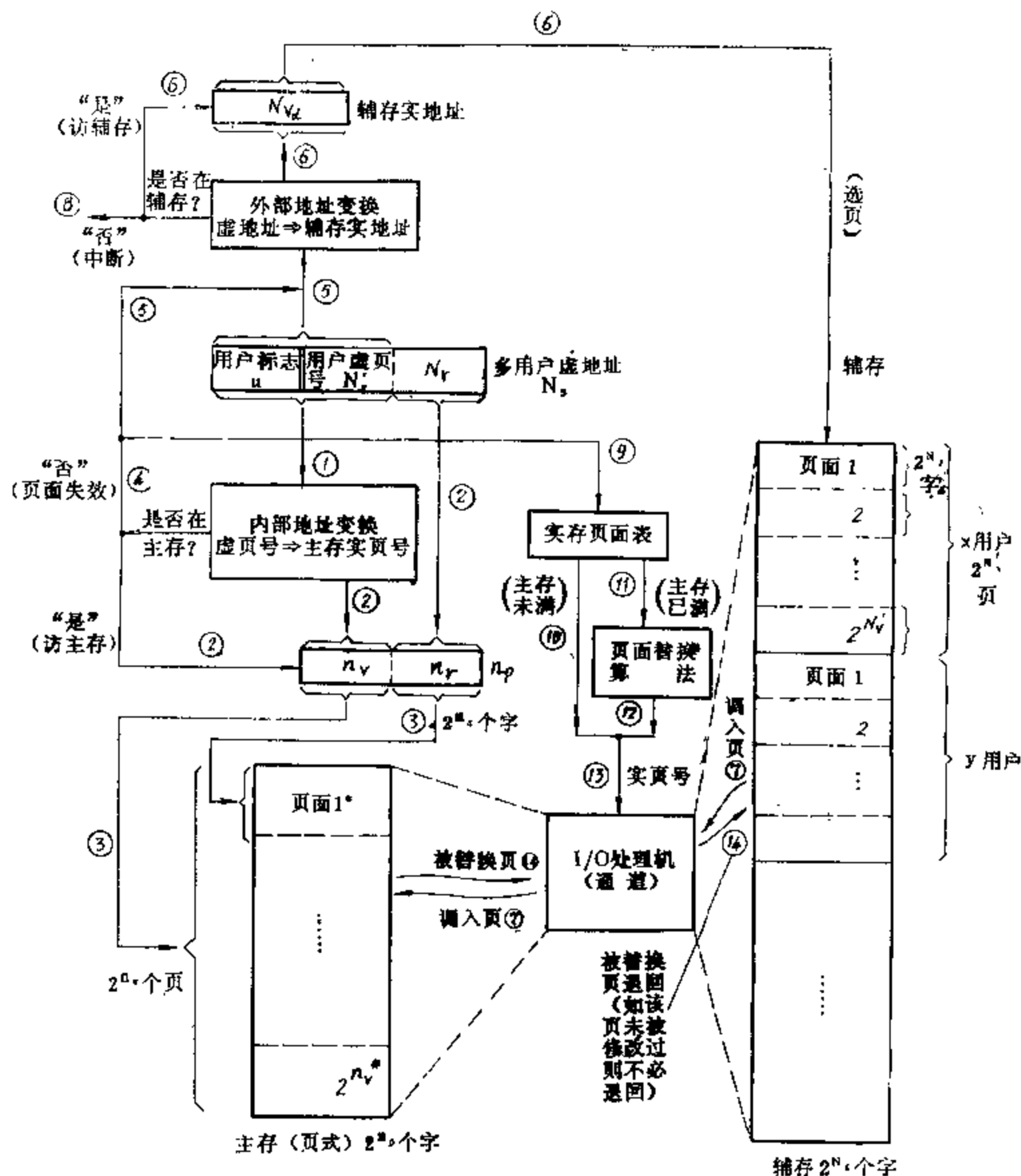


图 5.55 多用户虚拟存储器工作过程图

页表，将多用户虚地址变换成辅存中的实地址 N_{v_d} ，到辅存中去选页⑥，而后经过 I/O 处理机送入主存⑦。如果所需的页未装入辅存（即外页表中对应该页的装入位为“0”），还需再进入中断，由海量存储器调入⑧。在多道程序机器中总是希望 CPU 的运行与调页过程

(多用户虚地址变换到辅存实地址及把辅存该页传送到主存)并行进行,所以从辅存调该页进入主存是由能和CPU并行工作的I/O处理机(通道)来完成的。

在发生页面失效时还需要确定调入页应该进入主存中哪一个页面位置,这就需要查实存页面表⑨。这里有两种可能。当主存未装满时,只需找到空页(对全相联可进入主存中任意页)⑩;而当主存已装满时,就需要通过替换算法寻找替换页⑪⑫。对这二种情况,把确定的实页号送入通道⑬。在进行页面替换时,如果被替换的页调入主存后一直未经修改则不需送回辅存;如果已经修改,则需先将它送回辅存原来位置⑭,而后再把调入页装入主存⑦,是否修改过是可以由主存页面表指明的。

对页面失效的处理是设计存贮体系的关键之一,如果考虑不周,甚至会造成没法正确工作的局面。为此,我们在这里多讲几句。

我们在前面已经讲过,页面的划分只是机械地对虚、实存空间进行等分,与程序的逻辑结构毫无联系。这样,对于是按字节编址的,就有可能出现一条指令跨页存贮的情况;同理也有可能出现一个操作数跨页存贮的情况;而对于字符串,跨页存贮的可能性就愈大。还有,对于有间接编址方式的,尤其是有多重间接编址方式的,在间接寻址过程中,也完全可能会跨页(甚至跨多页)访问。因此,页面失效中断完全可能出现在取指令、取操作数、间接寻址等的过程中;就是说,页面失效中断是会在一条指令的分析、执行过程中发出,而且必须立即予以响应和处理,否则该条指令是没法进行下去的。这就破坏了第四章“中断系统”中讲过的,应在一条指令执行完了才响应中断的约定。

由此,就引起了在处理页面失效中断后,如何恢复中断现场,回到原中断点的问题,这当然需要软、硬结合地进行。有的机器的操作系统和存贮体系设计,就因为在改用虚拟存贮技术后,没有很仔细地考虑到这点,致使系统没法正确工作,从而需重新设计。

目前,不同机器有各种方法处理这个问题。有的机器是设计成在执行指令中间,出现页面失效中断时,采用后援寄存器技术把该条指令的现场全部保存下来;且当处理完该中断,把所需页调进主存后,能由该指令出现页面失效点处继续执行完该条指令。有的机器则是设计成在页面失效中断时的现场保存下来后,能使该条指令从头再执行。有的机器则是设计成在执行字符串指令前,予判字符串操作数的首、尾字符所在页是否已在主存内;只要有一个还没装入主存,就需发页面失效中断,在把该页调入后,才开始执行这条字符串指令;当然,这种方法只适用于字符串长度不得超过一页长的情况。

从前面的分析中还可看出,对跨A、B二页存贮的指令、操作数、字符串等,当B页还不主存,需调入时,替换算法的设计要做到不能正好把A页替换出去,不然就会出现前述的“颠簸”现象。若遇到指令和二操作数都是跨页存贮的最坏情况(其概率当然是很低的),则这条指令的执行就要求用到六个实页;因此,分配给各道程序的实页数还应考虑到上述情况。

由上述虚拟存贮器工作的全过程可以看出,它有二种地址变换。一种是由虚地址变换成主存实地址,它是每访主存一次就要进行一次,因此这种内部地址变换要求变换速度很高。另一种是只有当产生页面失效时才需进行的虚地址到辅存实地址的变换,这种外部地址变换如前所述速度可低。至于替换算法的实现,由于页面替换仅在页面失效且主存已满时才有,它的出现概率比外部地址变换还要低,所以,对它的速度要求可更低。

这样,就虚拟存贮器的速度要求来讲,关键是虚地址到主存实地址的变换速度。这个速

度如果达不到要求，虚拟存贮器就不能被采用。如何从逻辑结构上提高这个速度正是系统结构设计者的任务。

我们先分析前面讲过的页表法和目录表法是否能够满足速度要求。对页表法，因为页表是存放在主存中，因此每访主存一次，为查页表还需再加一次访存。如果采用段页式，仅查表就需访存二次；因此为存、取一个字，就需访存三次，比不采用虚拟存贮的多了二次，这会使速度严重下降，显然是不允许的。而相联目录表法，因为需要对 2^{n_v} 行进行相联查找（全相联时），这也是很费时间的。这样，就需要寻找别的途径来提高虚地址到主存实地址的变换速度。

5.1-4 快表与慢表

仔细分析实际查表的过程，可以发现，由于程序局部性的特点，对表内各行的使用不是随机性的，而是簇聚性的；就是说，不论是页表法还是相联目录表法，实际上在一段时期内只用到表中很少几行。这样，就可想到，能否不采用有 2^{n_v} 行的全“相联目录表”，而是用快速硬件构成比全表小得多的部分“相联目录表”，例如只有（8~16）行，那么其相联查找时间必然会加快。我们把这个部分目录表称为快表，只是当从这个快表查不到时，我们才从容量为 2^{n_v} 的内页表中找出对应的实页号，这个全表称为慢表，慢表存在主存中。这样，从虚地址到主存实地址的变换可以用图 5.56 的方法实现。

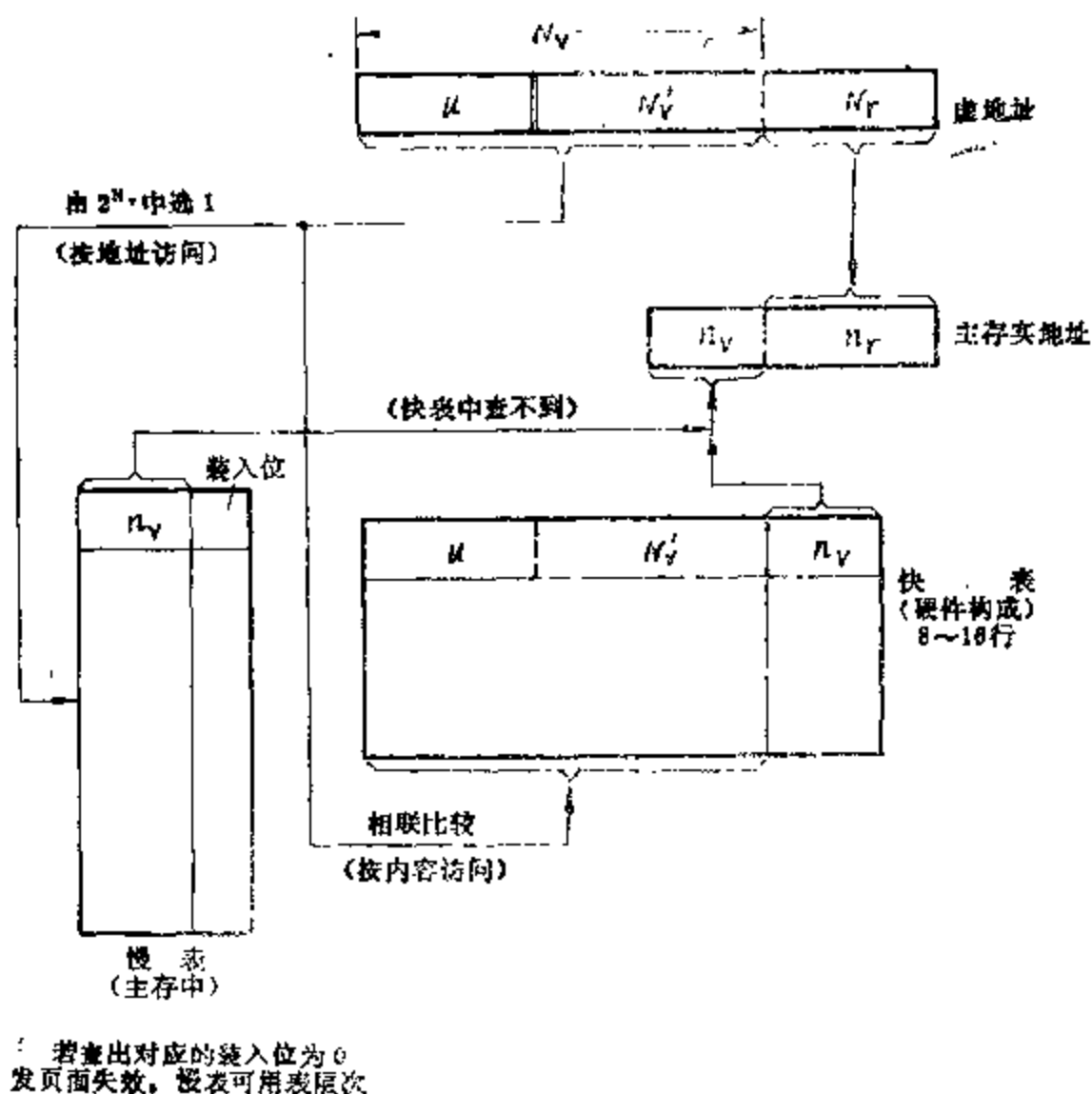


图 5.56 经快表与慢表实现内部地址变换

快表只是慢表的小小的复本，这样实主存地址的获得是来自二个途径。查表时，由虚页号 ($u + N_v'$) 同时去查快表和慢表。当在快表中有此虚页号时，就能很快地找到对应的实

页号 n_v 送入主存实地址寄存器，并使慢表的查找作废。从而就能做到虽采用虚拟存贮器但访主存速度几乎没有下降。如果在快表中查不到时，那就要费一个访主存时间查慢表。从中查到实页号 n_v 送入主存实地址寄存器，并将此虚页号和对应的实页号送入快表，替换快表中该移掉的内容，这要用到前述的替换算法。

虚拟存贮器的想法是在五十年代末就有了，只是在有了这种快表概念，并能以具体器件按速度要求实现之后才得以真正实用。当然，如果快表命中率不高，系统效率就会大大降低。如果快表的替换算法是前述堆栈型的，则快表容量愈大，其命中率必愈高，但容量愈大，其相联查找的速度也愈慢。所以快表的命中率和查表速度是有矛盾的。一般机器的快表取 (8~16) 行，若每页容量为 (1~4) K 字，则快表容量可反映主存中的 (8~64) K 单元，一般来说，这样的快表，其命中率是不会低的。

可以看出，快表与慢表实际上构成了层次，它与主存—辅存层次在概念上是相同的，因而前述替换算法等也适用于它。快表—慢表层次的替换算法一般也采用 LRU 法。

为了使快表的查找尽可能快，还可以在结构上采取某些技巧。例如，在相联比较的位数上动脑子，因为对一般器件，在同样容量情况下，相联比较的位数越少，则相联查找所花费的时间和设备量就越少。因为快表内容在一段时间内总是对应于同一个任务，同一个用户，就是说它们的 u 值是相同的，所以参与相联比较的位数中就可把 u 去掉，这就可以由 N_v (即 $u + N_v'$) 位缩短为 N_v' 位，如图 5.57 所示。

然而，这样一来却会引出新的问题。显然，快表应对系统程序员是透明的（更不用说对应用程序员了）。“透明性”是系统结构对软件的重要支持，是软件所希望的。然而，操作系统设计者和系统结构设计者都必须对包括快表的这些“透明性硬件”进行慎重的分析，看看是否在任何情况下都不会使系统出错，而且不会对速度等性能有不利的影响；尤其要分析

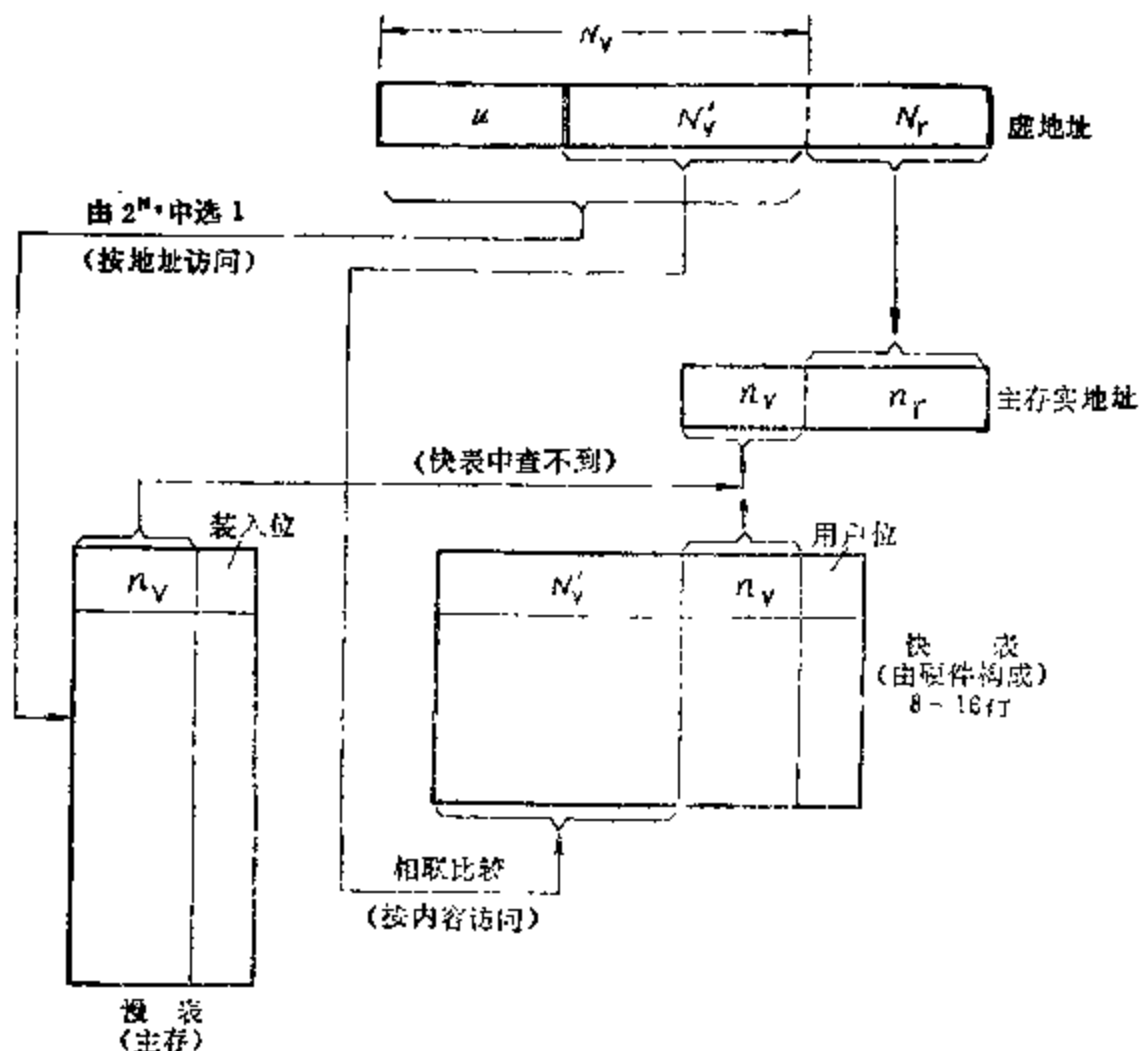


图 5.57 减少快表的相联比较位数

在任务（进程）切换时的情况，因为“透明性硬件”既是“透明”的，那就是系统软件所控制不了的。

对于上述把相联比较的位数由 $u + N_v'$ 位缩短为 N_v' 位，在任务切换时就会出错，因为在 u 值改变后，同一个 N_v' 值所对应的 n_v 值当然是不同的。为了使快表在相联比较位数缩短后仍能反映出这点，早期的快表为此设置了一位用户位（见图5.57），用户位“0”的行，其内容无效。在任务切换时，通过硬件（如设计一条专用指令）将所有用户位全部置成“0”，这就相当于对应上一个任务的快表内容全部作废，这时只能通过慢表取得 n_v 值，在装入快表的同时，将该装入行的用户位置成“1”。用这种方法来保证快表中有效的内容是对应于现行任务的。

然而，这种方法却可能损失虚、实地址的变换速度，尤其在快表容量增大时更是如此。例如，当某个 A 任务要使用 I/O 设备，它是通过“进管”指令进中断调用操作系统，这时就要进行任务切换，要使快表中的所有用户位都置成“0”。可是，对于有 I/O 处理机（通道）的机器，操作系统为启动 I/O 设备只需进行为数不多的操作，因此，CPU 很快就可返回运行 A 任务。但是，因为对应 A 任务的快表内容已全部作废，所以使得在相当长的一段时间内，虚、实地址的变换速度因需查慢表而显著下降。显然，这是由于快表内容不恰当地被全部作废所引起的。

为了解决这个问题，有的机器就把前述专用指令的功能改为只清除快表中指定行的使用位，从而可使其它行的内容在任务切换回来后仍然有效。然而，不论专用指令是清除整个表的，还是只一行的使用位，都使得快表对系统程序员不是透明的；这就要增加操作系统的负担，当然不是理想的解决办法。

那么，能否做到既能减少快表的相联比较位数，又能在任务切换时不会出现混乱和错误，而且还能使快表对系统程序员确是透明的呢？IBM370/168 比较好地实现了这点，它的解决方法下面再讲。之前，我们先讲讲增大快表容量的办法。

前面讲过，快表容量越大，命中率就越高，但相联比较却越费时，导致快表快不起来。解决这个矛盾的途径之一是下用相联存贮器而是用速度快得多的按地址访问存贮器片子来构成容量更大的快表，并采用 § 3.6 所述的用硬件实现的散列法，使快表的地址 $A = H(N_v)$ 。如图 5.58 所示。

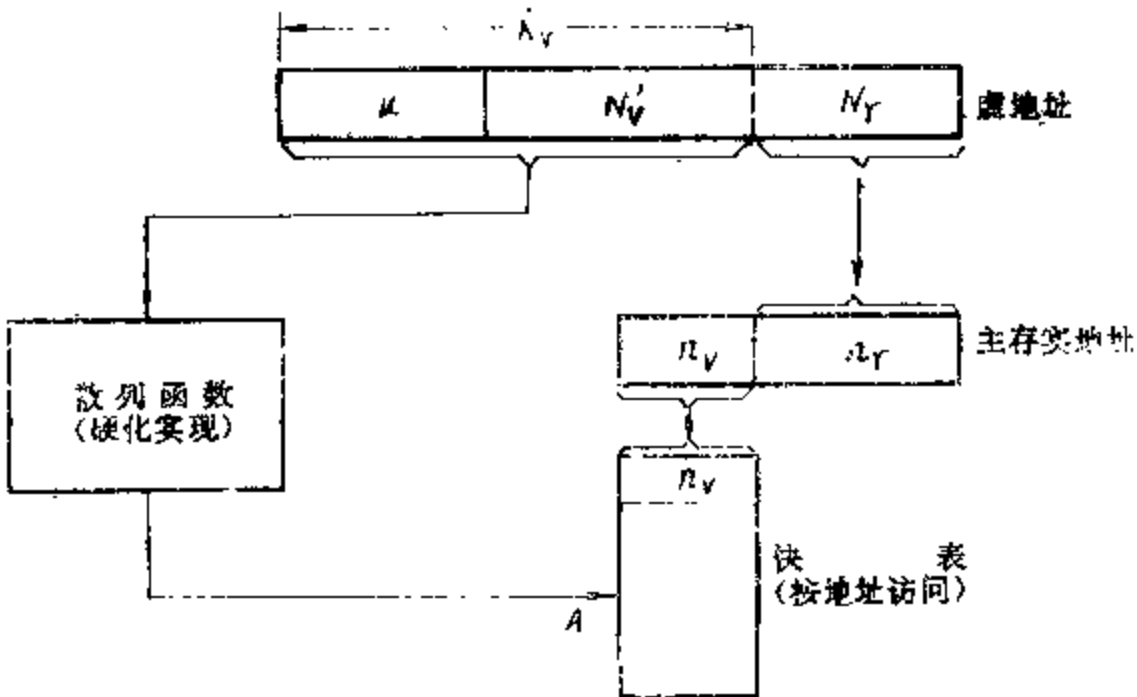


图 5.58 经散列实现快表按地址访问

然而，散列不是唯一的，就是说 A 与 N_v 不是一一对应的，存在可能查错的问题。虽然快表的快是前提，但总得保证准确，为此需在快表内加上 N_v 项，使之在用散列法求得 A 查到 n_v 后，将同行的 N_v 与虚地址中的实际 N_v 作相等比较。若不等就表明出现了散列冲突，即 A 地址单元的 n_v 不是虚地址对应的实页号，这时就从查慢表来获得 n_v 。这里的判相等可与访主存重迭进行。这样构成的快表，其容量可比前述相联查找法的 8~16 行大，达 64~128 行，从而能进一步提高快表的命中率，而仍能有很高的查表速度，其原理图见图 5.59。

采用散列方法，可以把有几十位宽的 N_v 压缩成几位（例如 6~7 位）宽的 A 。

综上所述，这种方法的出发点是让概率高的操作尽可能用最快的方法处理，那怕可能不准确或错了，只要出错概率很低且能发现和纠正，甚至这种纠正要花费较多的时间也仍然会带来系统效率的提高。这是设计当代机器的重要思路，不只是可用于存储体系的设计，它可以看成是前述 Huffman 概念的一种应用。

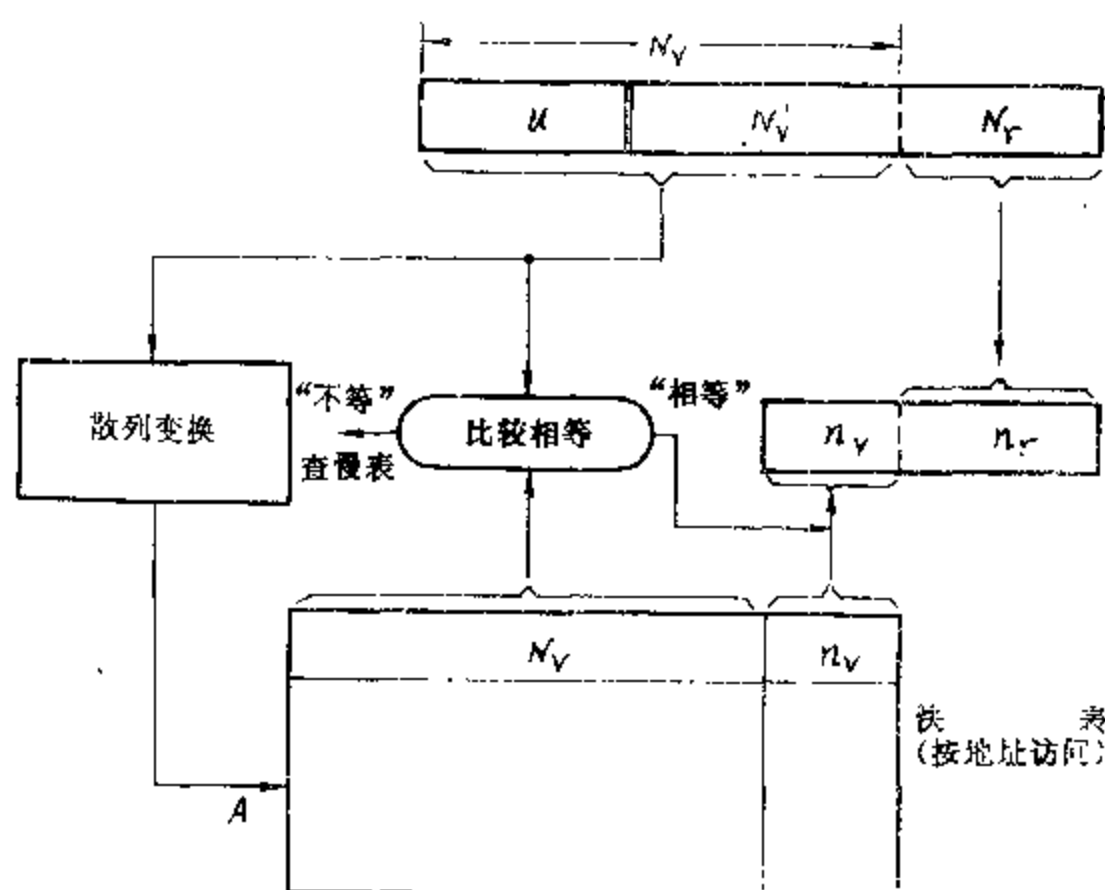


图 5.59 快表中增加 N_v 比较以解决散列冲突引起的查错

如果给上述快表再加某些措施就可降低由于散列冲突所引起的不命中率，IBM370/168 的虚拟存储器就采用这种办法，见图 5.60（图中只画出快表部分）。它使每个地址 A 对应二个虚页号，相当于把 § 3.6 的 $H(\text{Key}_1)$, $H(\text{Key}_2)$ 放在同一个 A 单元内，用两套相等比较电路，哪一个相符就送哪一个的 n_v ，只有当 A 单元的二个虚页号都不相符时，才是不命中，才需查慢表来获得 n_v 。

此外，散列变换（压缩）的（入、出）位数差愈小，散列冲突的概率就愈低，因此需要缩短被变换的虚地址位数。然而前面讲过，要把 u 位去掉是不行的；但是 370/168 的设计者考虑到这 24 位长的 u 可对应于 2^{24} 个任务，但在比较长的一段时期内可能仅几个在运行，远比 2^{24} 小得多，换句话说，在同一时期内 u 的变化概率比起 N_v' 的变化概率要小得多，因此，只需把这几个 u 值存在例如六个相联寄存器内，从而只需用三位 ID 表示，相当于经过只有六行的高速相联存储器把 24 位 u 压缩成三位 ID。

在进行地址变换时，先将虚地址中的 u 在这相联寄存器组中进行查找。找到相应的 ID

值（三位），再与 N_v' （12 位）拼接，使得需相联比较的位数由 $u + N_v' = 32$ 位缩短成 $ID +$

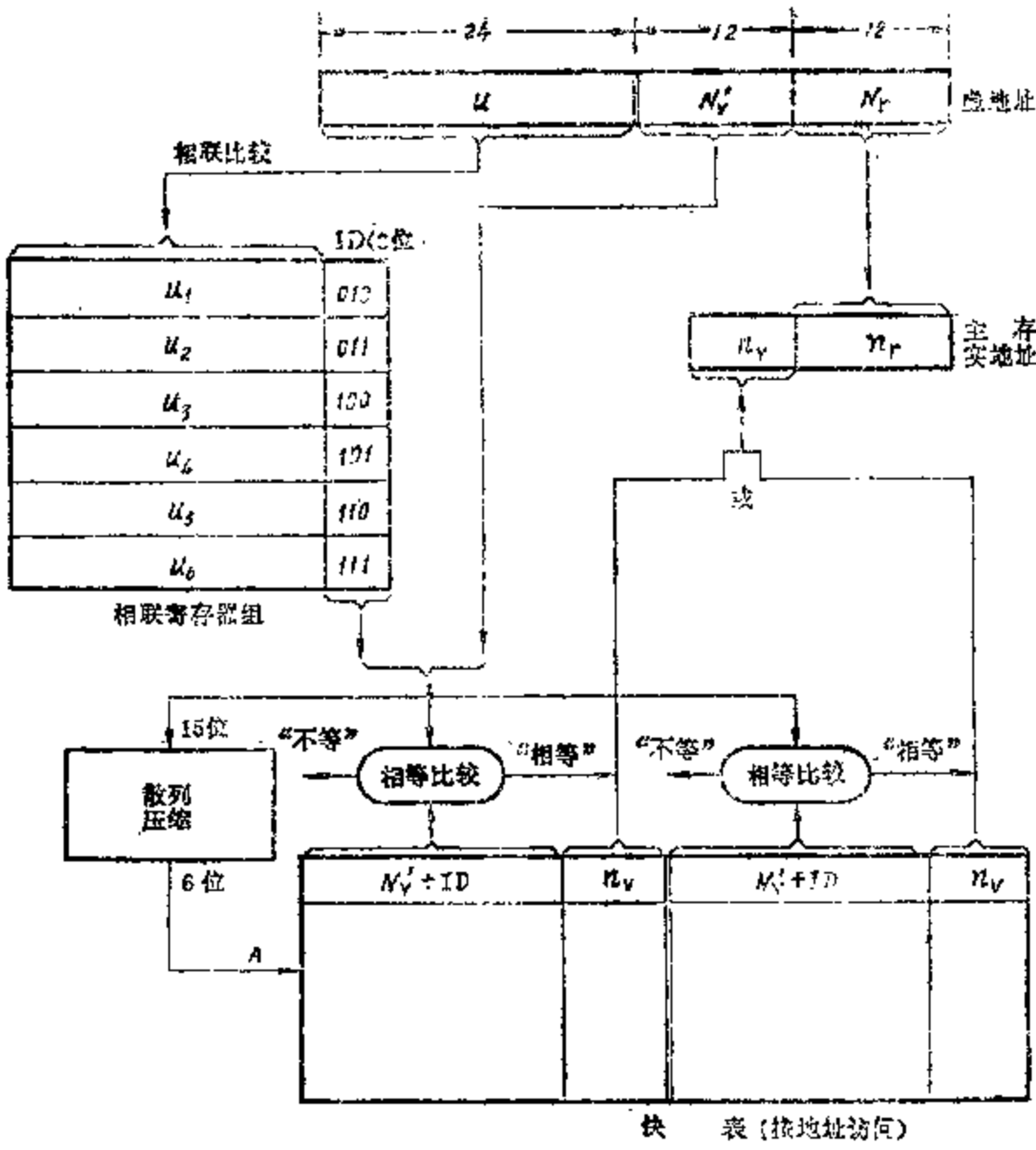


图 5.60 IBM370/168 虚拟存储器快表示意图

$N_v' = 15$ 位。从而，既能缩短相联比较的位数，缩小散列变换的入、出位数差，又仍然能达到在任务切换时不会出错。这样，在快表内可以同时存有至多六个任务的部分页表；而且，在任务切换时，不必由操作系统使整个快表或其某行的内容作废，即快表对操作系统，对系统程序员是真正的“透明”。只当某个 u 值与六行相联寄存器组的哪一行都不相符时，表明已切换到这六个任务之外的新任务，才需按前述替换算法逐行地替换进该新任务的页表内容。看出，快表内容是随着任务的切换逐行地自动更换；而且常用任务（或主任务）的那部分页表内容被替换出去的概率很低，从而解决了前述某个 A 任务为调用 I/O 设备，切换到操作系统，而又返回 A 任务后，因快表内容作废所造成的虚、实地址变换速度下降的问题。在 370/168 之后的机器，它们的快表结构就基本上与 370/168 的相似。

举上述应用例子是想说明把软件概念（如散列技术）应用于系统结构设计所带来的好处以及软硬相互渗透，努力设计出好的“透明”硬件对提高计算机系统性能的好处。

5.2 影响虚拟存储器某些指标的因素

本节简要分析影响主存利用率和命中率的一些因素。

5.2-1 主存空间利用率

由于价格的限制，主存容量是有限的，如何减少主存的浪费，提高其利用率是设计存贮体系时必须考虑的。就每道程序来看，主存空间利用率可粗略地定义为被程序的“活跃”部分占据的存贮空间与分配给该道程序的主存容量之比，它当然是越接近于1越好。

在§2已讲过另头的浪费；另外存在主存的映象变换表（如段、页表）也要占去一部分空间。还有，程序中有些部分例如诊断程序、出错处理程序等等是调入主存后可能从没用过就调回辅存的，这部分非活跃程序所占用的存贮空间也可以说是浪费的，如果程序的定位算法能更好地反映程序的局部性，这些浪费可以减少。下面分析页式管理的另头、页表大小与主存空间利用率的关系。

设 S_s 为程序的平均长度（按字编址），其最末一页会有页内另头浪费，当页面大小 $S_p \ll S_s$ 时，此另头的平均值可认为是 $\frac{S_p}{2}$ 个字。看出， S_p 越大，内另头损失越大，从这个观点来看，是宜于选用小的页面以提高利用率。然而，程序需有页表，其大小为 S_s/S_p 个字（假定页表宽度为单字长，整个页表存于主存），若页面过小，就需用很大的页表，这也会降低主存空间利用率。因此，页面过大过小，都会降低主存的利用率，所以，在虚拟存贮器刚出现时，有人就企图从如何能使主存利用率最高来寻找最佳页面大小。由于程序的页内另头和页表共耗费存贮空间

$$S_u = S_p/2 + S_s/S_p \quad (5.5-1)$$

则对应该程序的主存空间利用率

$$u = \frac{S_s}{S_s + S_u} = \frac{2S_s S_p}{S_p^2 + 2S_s(1 + S_p)} \quad (5.5-2)$$

u 的最大值应对应于 S_u 的最小值，而对应最小 S_u 的 S_p 就是最佳页面大小 S_p^{OPT} ，所以对 (5.5-1) 式求最小值：

$$\frac{dS_u}{dS_p} = -\frac{1}{2} - \frac{S_s}{S_p^2} = 0$$

$$\text{得} \quad S_p^{OPT} = \sqrt{2S_s} \quad (5.5-3)$$

$$\text{这样，} \quad u^{OPT} = \frac{1}{1 + \sqrt{2/S_s}} \quad (5.5-4)$$

图5.61为不同 S_p 值的 u 与 S_s 的关系曲线。看出，如果只从主存空间利用率考虑，页面大小应按 $\sqrt{2S_s}$ 选取，但这个值往往比实际机器所用 S_p 值要小。例如，对于 $S_s = 5K$ 字， $S_p^{OPT} = 100$ 字，仅当 $S_s = 500K$ 字时， $S_p^{OPT} = 1K$ 字，但后来的实际机器在主存容量为 500K 字以下，且分配给每道程序的空间比 500K 字小很多时，其 S_p 值也取为 1K 字。原因就在于采用前述的表层次技术，并不需要把整个页表存在主存内，而且 S_p 的确定不能只从主存空间利用率这一点考虑，

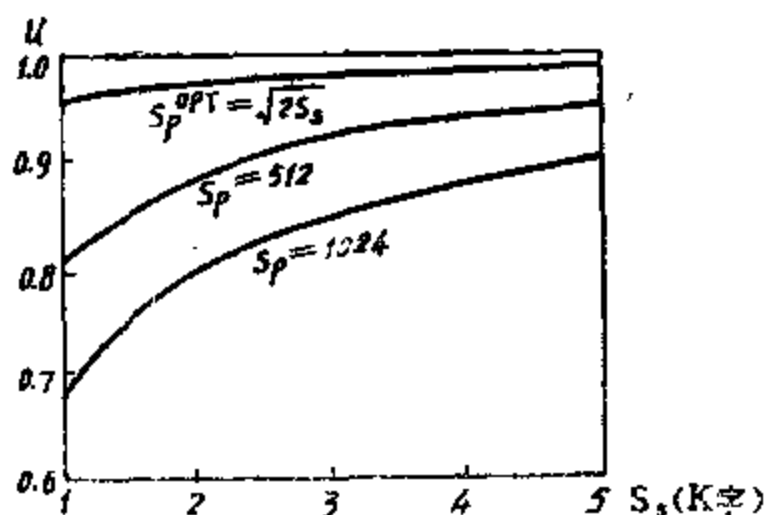


图 5.61 对不同页面大小 S_p ，主存空间利用率 u 和程序大小 S_s 的关系曲线

还应考虑其它的因素。例如，对磁盘辅存，其查找时间要比传送时间长得多，从这点出发那是希望页面要大，因为不论页面多大，其查找时间都一样，所以页面愈大，调页效率愈高；又如，还应从出现前述的指令、操作数和字符串跨页存贮的概率来考虑页面的大小；还有，还应考虑到操作系统为页面替换所需的辅助操作有多少，耗费时间有多长。此外，页面大小还应联系下述提高命中率的需要来确定。

5.2-2 主存的命中率

前面讲过，命中率是评价存贮体系的重要指标，而页面大小 S_p 和 分配给该道程序的主存容量 S_1 是影响命中率 H 的重要因素。其典型关系可如图 5.62 所示。

对给定 S_1 ，随着 S_p 的增大， H 先是增大，达到某个最大值后，随 S_p 增大反而减小。

原因是这样的，设地址流 A 中相邻逻辑地址之间的距离为 d ，若 d 比 S_p 小，则是同页内的访问，当然是命中，从这点看， H 是随 S_p 的增大而上升，而在 d 比 S_p 大时，若二个地址相隔较远，且不属同一页，但该地址所在页也已在主存中，那也是命中，从这点看， H 是

会随分配给该道程序的实页数的增加而上升，在 § 4 已证明，对于堆栈型替换算法这是必然的。当增大页面时，对于同页内的访问，当然是使命中率上升，然而对二个地址分属不同页的情况，若分配给该道程序的主存容量是固定的，则总页数要减少，因此就可能使命中率下降。当 S_p 较小时，在增大 S_p 的过程中，前一种因素起主要的作用，因此 H 随 S_p 的增加而上升，达到某个最大值之后，后一种因素起主要作用，因为页数要明显减少，这就导致增大 S_p 反而使 H 下降，而且偶然性的访问某些页的页面失效概率也上升。当然，从图上可见，增大分配给该道程序的容量 S_1 ，能延缓由于上述后一种因素所引起的 H 下降。

然而，增大 S_1 也是在开始时作用明显，如图 5.63 所示。而且，从主存空间的利用率看， S_1 的增大可能由于不活跃部分所占比例增大而使其利用率 u 下降。若 S_1 值取得能把整道程序都装入，那当然命中率最高，但同一时期只有一部分在用，必然使主存利用率变坏，而且这也不符合采用存贮层次的目的。因此， S_p 、 S_1 的选择都是折衷平衡，目标是使系统的性能价格比尽可能高。可惜的是至今还没有确能反映实际的模型，以致不能定量地确定 S_p 值。下面是模拟法的某些结果。

对某个 FORTRAN 作业流，使用 LRU 替换算法和全相联映象，采用不同页面大小模拟出的不命中率（即页面失效率 $= 1 - H$ ）与 S_1 的关系，如图 5.64 所示。看出，对于不命中率在 1% 以下，曲线在 S_p 增加到几乎是 4K 时都是一直减小的，而之后才开始增大，这和

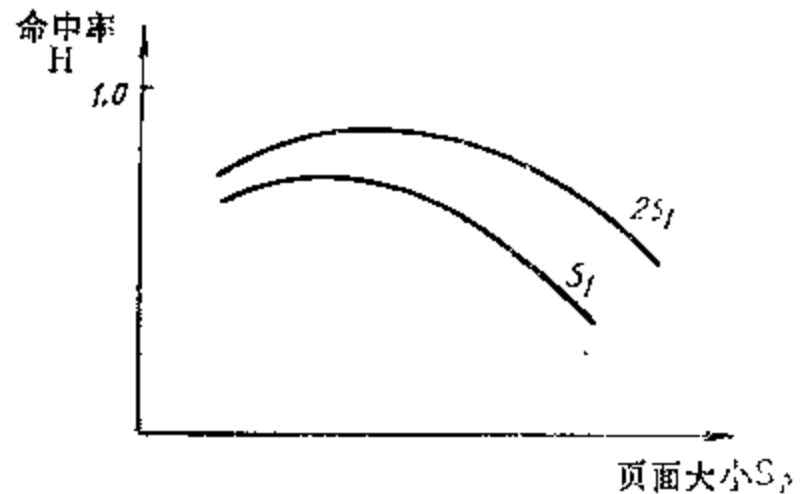


图 5.62 页面大小 S_p 、容量 S_1 与命中率 H 的关系

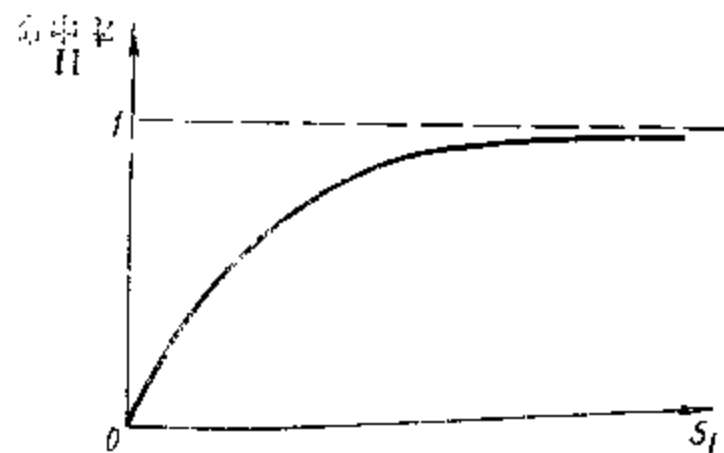


图 5.63 H 与 S_1 的关系

图 5.62 的分析是相符的, 从这点出发, 页面大小的最佳值为 1K 到 4K 左右。它比按主存利用率考虑的最佳页面大得多。就虚拟存贮器的命中率和主存利用率这两个指标来看, 命中率是更重要的, 所以页面大小应按命中率选定, 但要考虑到不要使使用率过低。目前一般机器的页面大小就是取为 1K~4K 字节。

除了页面大小和分配给程序的主存容量外, 替换算法也对命中率有重要影响, 这在替换算法那节已经分析过了。

在以上的讨论中, 虽然联系了多道程序的情况, 然而主要是围绕单道运行的情况来分析。从多道运行看, 影响的因素就要更多。

例如, 对分时系统, 分配给每道程序的 CPU 时间大小会影响对虚拟存贮器的使用。

如果分配给的 CPU 时间较小, 就应尽量减少页面交换的次数, 不然所给时间的大部分就会被调页时间消耗掉; 然而对 S_1 的要求却可降低, 因为在小的 CPU 时间内, 来得及使用的主存容量会较少。同理, 页面也不能选的过大, 不然会出现连一页也没用完, 就得换道了。

前已说过, 在多道程序时, 每道程序占有的主存页面数比之只运行单道程序时所具有的当然要少, 这会使该道程序的主存命中率下降。从而对该道程序来讲, CPU 效率是下降了。但是, 另一方面, 由于这时 I/O 和 CPU 是重迭工作, 使得在不命中而需访辅存调页时, CPU 不是空等, 而是换到另一道程序; 因此对 CPU 来讲, 多道时的效率会比单道的高。那么, 道数增到多少时会出现由于命中率降低引起的效率降低比因为由辅存调页时, CPU 可以换道所带来的效率的增加来得更大呢? 有人用概率模型进行模拟分析, 认为在共同运行的作业数 J 小于某个 J_{opt} 值时, 因为调页和 CPU 重迭工作而使 CPU 效率随 J 数的增大而增加, 但当 $J > J_{opt}$ 后, J 的增加会使每个作业的主存分配量过小而使主存命中率过分下降, 从而使 CPU 效率下降。

已提出了在多道程序系统中优化 CPU 效率的各种调页模型。

一种认为应遵循所谓 50% 准则, 即如果调页操作能使辅存约有一半时间是忙着的, 则 CPU 的利用率视为最大。这里多道程序的道数及随之而来的为每道程序的主存分配量明显影响页面故障率。

另一种认为当调页时间近似等于页面故障间的平均时间时, CPU 利用率最高。

第三种认为每道程序的页面分配量应选择成能使页面故障间的平均时间达最大值。

按照这些见解就可提出调整道数 (并行作业数) 的算法以及使系统吞吐量最大的存贮管理策略。

关于虚拟存贮器的基本概念就讲到这里。在前面, 我们都是按虚存空间大于实存空间来讲述的, 这应该是虚拟存贮系统的基本特征。然而, 也有一些小型机是把小的虚存空间映象

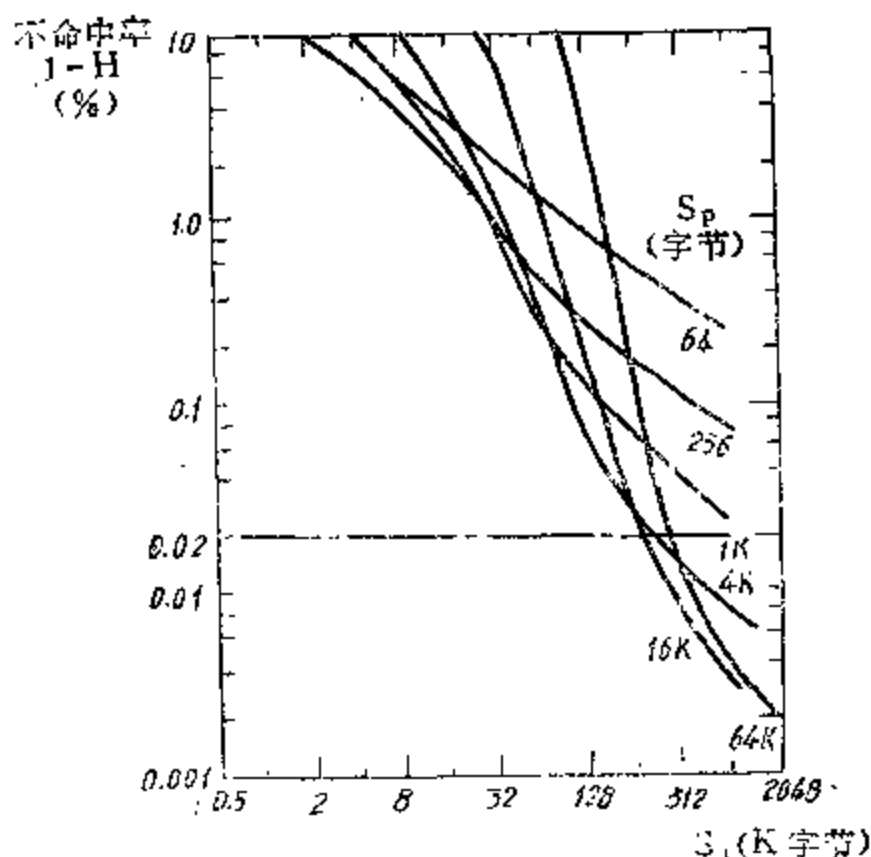


图 5.64 使用 LRU 替换算法, 全相联映象, 对于不同 S_p , 某个 FORTRAN 作业流的不命中率与 S_1 的关系

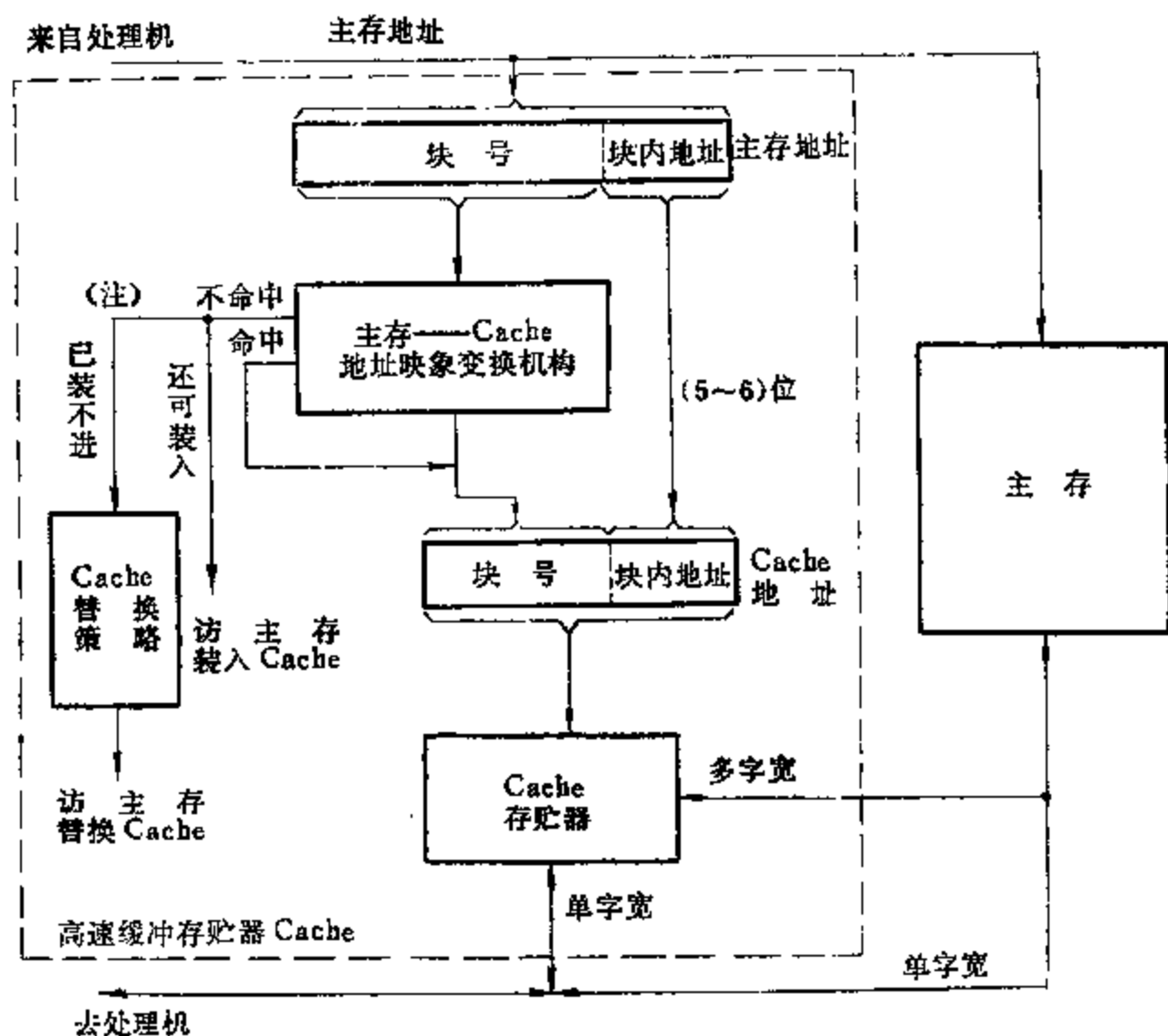
到大的实存空间，如 PDP-11 就是这样，它的虚地址是 16 位长，可变换到 18 位长的主存实地址。这样，就能使得机器的实际主存空间可大于原先机器地址码所能表示的最大地址空间，是扩大机器存贮功能的一种办法。78 年问世的 IBM SERIES/1 机器也是这样，它的按字节编址的虚地址为 16 位长，页面大小为 2K 字节，这相当于 $N_v = 5$ ， $N_r = 11$ ；但其主存实地址可达 24 位长，相当于 $n_v = 13$ ， $n_r = 11$ 。这样，它的页表只需有 $2^{N_v} = 32$ 个入口，从而可用高速随机访问存贮片子构成，以实现快速的虚、实地址变换。

§ 6 Cache 存贮器

6.1 基本结构

在前面 § 1.2 节已讲过，构成“Cache—主存”层次的目的使该层次既具有 Cache 的速度，又具有主存的容量。

工作原理如图 5.65 所示。



(注) 对组相联，装不进不等于 Cache 已满。

图 5.65 Cache 的工作原理图

Cache 和主存都分成块(行)，每块(行)一般是 8~16 字 (32~64 字节)，由主存地址通过主存—Cache 地址映象变换机构判定是否在 Cache 中，并变换成 Cache 地址。如果它不在 Cache 中就产生 Cache 块失效 (Cache 不命中)，这时，就要访主存调块，若装不进 Cache，则需按某种替换策略，把该块替换进 Cache。显然，只要 Cache 命中率足够高，就相当于能按 Cache 的速度向处理机提供指令和数据信息。

对单处理机系统，采用 Cache 后可以降低对主存的速度要求；而对共用主存的多处理机系统，如果每个处理机都有它自己的 Cache，则处理机主要与 Cache 打交道，能大大减少主存使用的冲突，提高整个系统的吞吐量。一般是尽量使处理机与 Cache 在位置上相近，使之能更好地发挥 Cache 的高速性。

加 Cache 后所需增加的成本，可以因为能降低对主存速度的要求，从而降低主存的造价来补偿，使得机器的性能价格比显著提高。

目前 Cache 不仅是大型机上普遍采用，小型机上也使用，如 PDP-11/44 和 IBM 4341 都有 8 K 字节的 Cache。

“Cache—主存”存贮层次和“主存—辅存”存贮层次在概念上是一样的。因而，前述地址映象和变换以及替换算法等的分析对“Cache—主存”层次也适用，只是现在虚地址指的是主存地址，实地址指的是 Cache 地址。下面主要分析“Cache—主存”层次与“主存—辅存”存贮层次的不同点。

第一，在实现方法上的不同

一般要求 Cache 系统的访问时间等于处理机拍宽。而且，Cache 与主存的访问时间比一般不到 1/10，而主存与辅存的访问时间比一般达 1/1000 以上。由于操作速度的要求，

“Cache—主存”层次不仅是要求地址变换全用硬化快表实现，而且替换算法也是硬化实现。如采用在 § 4 中讲过的堆栈法和比较对法等。既然地址变换全用硬化目录表实现，那若采用全相联映象，则快表的容量将过大，以致快不起来，所以通常采用 § 3.3 讲过的组相联映象以缩小快表容量，尽管这样会使块冲突的概率有所增加。

由前面图 5.65 看出，从送入主存地址到 Cache 输出，包括地址变换和 Cache 存贮器的读出二部分时间。由于地址变换需访问组相联块目录表，其所需时间可能和 Cache 存贮器的读出时间相近。前面说过，要求这二个时间的和应等于处理机拍宽，若达不到，可用图 5.66 的流水方法解决。看出，虽然从送入地址到取得数据需 160 毫微秒，比拍宽大一倍，但经流

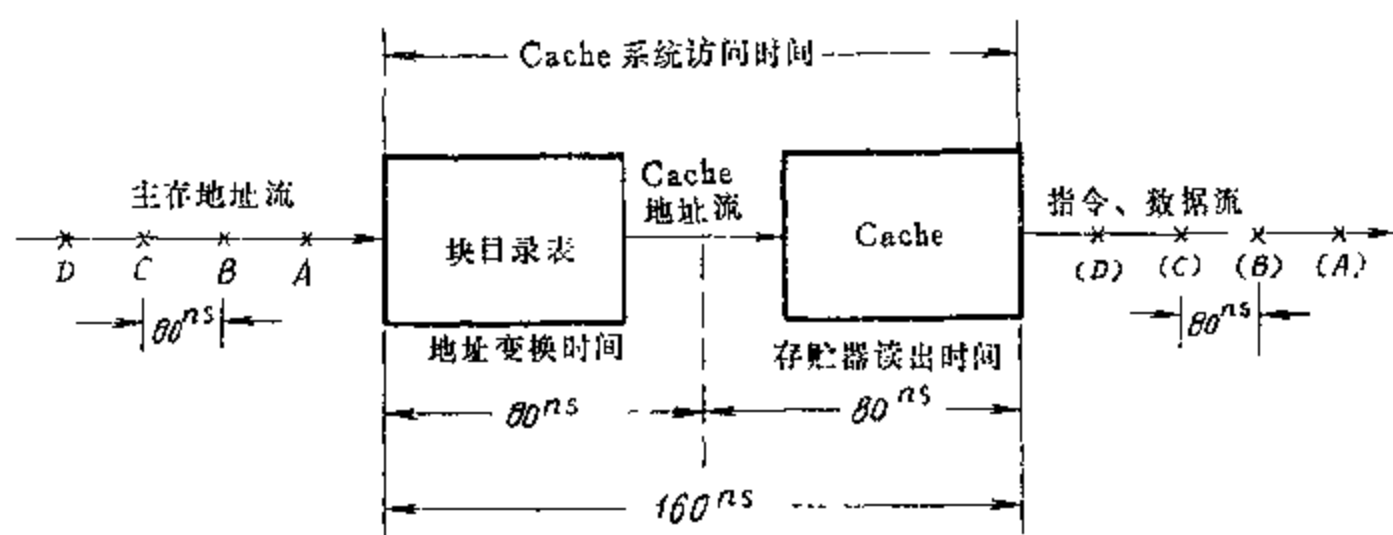


图 5.66 用流水技术提高 Cache 的速度

水，每隔一个拍宽 80 毫微秒仍可以从 Cache 中取得一个数据。

第二，在处理机和 Cache、主存的联系上

在虚拟存贮器中，处理机与辅存之间没有直接的通路，因为辅存的速度相对主存的差距很大，因此一旦发生页面失效，由辅存调页的时间是毫秒级，为使处理机在这段时间内不致白等，前面讲过，处理机一般是采取换道到其它程序的办法。而对于 Cache 存贮器，当

Cache 页面失效时,不能采取换道的办法,因为由主存调块的时间是微秒级,比换道的时间要短得多。为了减少处理机的等待时间,除 Cache 到处理机的通路外,还可增加主存直接到处理机的通路,如图 5.65 所示。因为 Cache 块失效时,处理机要的是一个字,因此不必等主存把整块调入 Cache 后再由 Cache 把所需的字送入处理机,而是使 Cache 调块与处理机访主存取所需字重迭进行。我们把主存到 CPU 的直接通路称为读直达通路。这样,Cache 既是“Cache—主存”层次中的一级,又是处理机与主存间的一个旁视存贮器。

其次,为了加快调块,在主存与 Cache 之间采用多体交叉多字联系(见 § 1.3 节)。每块的容量一般等于在一个主存周期内由主存所能访问到的字数。例如,370/168 的主存是模四交叉,每个分体是 8 个字节宽,所以 Cache 的每块容量为 $4 \times 8 = 32$ 个字节;CRAY—1 的主存是模 16 交叉,每个分体是单字宽,所以其指令 Cache 的块容量为 16 个字。

另外,主存是被机器的多个部件所共用,应把 Cache 的访主存优先级尽量提高,一般应高于通道的访主存级别。因为 Cache 的调块时间,只占用 1~2 个主存周期,这样做不会对外设访主存带来太大的影响。

第三,Cache 是透明的

既然 Cache 的地址变换和块替换算法的实现全是硬化实现,因此,“Cache—主存”层次对应用程序员和系统程序员应该都是透明的;而且 Cache 对处理机和主存之间的信息交往也是透明的。然而,正如在上一节对快表的透明性所进行的分析一样,对于 Cache 的这些透明性所可能引起的问题及其影响需要慎重分析。

虽然 Cache 是主存的部分付本,然而,对应同一个主存地址,在主存内该单元的内容和 Cache 内对应此主存地址的内容却可能在一段时期是不相同的。例如,当中央处理机往 Cache 写入,即修改 Cache 内容时,虽然 Cache 内的这部分内容已经变化,但主存中的这部分内容却可能仍然是原来的,还没变化。同样,当 I/O 处理机已把新的内容送入主存某个区域,但 Cache 内这个区域付本的内容却可能还是原来旧的,致使中央处理机由 Cache 读出错误的信息。然而,Cache 对处理机和主存都是透明的,因此上述读、写过程所引起的 Cache 的内容跟不上主存对应内容的变化和主存内容跟不上 Cache 内容的变化都是可能会引起错误的。下面先分析后一种跟不上所可能引起的问题,而后再分析如何处理前一种跟不上。

当中央处理机往 Cache 进行写入时,若主存中的对应部分仍是原来的,则对 Cache 的块替换和中央处理机与 I/O 处理机等其它处理机经主存交换信息时会造成错误。

我们先分析对块替换的影响及其解决办法。在前面讲过,对虚拟存贮器的页面替换,如果在主存中被替换的页面已被 CPU 修改过,但辅存中对应的该页内容仍是原来旧的,则需先把被替换页写回辅存,才能调入新页,这被称为写回法或抵触修改法。这种策略在 Cache 的块替换中也采用。但由于“Cache—主存”层次中还有主存到处理机的直接通路,因此还可以用写直达法。这种策略规定当处理机往 Cache 写入的同时也写入主存,从而使得主存内容始终跟得上 Cache 内容的变化。这样,当 Cache 中该“块”被替换时,就不必先写回主存,而是可以立即被替换。但是这种策略每次写 Cache 时都要附加比写 Cache 时间大得多的写主存的时间。

由于处理机的不少写入都是某个中间结果,往往在 Cache 中该“块”被替换之前还需

经多次修改，这种中间的多次修改是可以不必写入主存。由于 Cache 的命中率又较高，所以若只从处理好块替换，则 Cache 是可采用和虚拟存储器一样的写回法，而不采用写直达法，以减少把中间结果写入主存所耗费的时间，只有当 Cache 中被写入过的块要被替换时才一次写回到主存中。

但是，对于共用主存的多处理机系统（包括只有一台中央处理机和 I/O 处理机的系统），由于各处理机之间的交换信息是通过主存进行，若采用写回法，则只有当存有交换信息（例如是中央处理机要送往 I/O 处理机的信息）的块被替换，送回主存后，别的处理机（例如 I/O 处理机）才能取得正确的信息。因此，宜于采用写直达法。所以，具有 I/O 处理机（通道）的 IBM 370 系统就是采用写直达法。还有的机器，如 Amdahl 470/V 6 则是采用使 I/O 处理机直接与 Cache 相连，再经它接往主存，对 I/O 处理机来讲，Cache 是透明的。

至于写不命中后的处理策略可有二种。一种是“不按写分配”法，它是当 Cache “写不命中”时，只写入主存，但这个写地址单元所在的块不从主存调进 Cache；另一是“按写分配”法，它是当 Cache “写不命中”时，除写入主存外，这个写地址单元所在的块还由主存调进 Cache。这两种策略对不同的替换算法效果不同，但两者的差别不大。根据模拟，对于采用 LRU 算法，在 Cache 容量不是很大时，采用“不按写分配”法对提高命中率有好处。

至于 Cache 的内容跟不上已变化了的主存内容的问题，一种解决办法是当 I/O 处理机往主存写入（不是经 Cache）新内容的同时，由操作系统经某个专用指令清除整个 Cache。这种办法的缺点如我们在上一节讲述用专用指令清除快表一样，突出的是使得 Cache 对操作系统，对系统程序员不是透明的。另一种办法是当 I/O 处理机往主存某个区域写入新内容时，由专用硬件自动地将 Cache 内对应此区域的付本作废，而不必由操作系统进行任何干预，从而保持了 Cache 的透明性。

总之，系统结构设计者必须清楚地认识到 Cache 的透明性所可能引起的问题，并采取妥善办法来解决。

6.2 影响“Cache—主存”层次性能的因素

和虚拟存储器中的讨论类似，评价 Cache 存储器，主要是看命中率的高低，而命中率是与块的大小、块的总数（即 Cache 的总容量）、采用组相联时的组的大小（组内块数）、替换策略和地址流情况（如簇聚性状况）等有关。

这些因素对命中率的影响可以通过模拟来寻求优化参数。不命中率与 Cache 的大小、组的大小和块的大小关系一般如图 5.67 所示，可见块的大小，组的大小及 Cache 容量这三者的增大都会提高命中率。因为 Cache 调块时，处理机是空等，所以要求调块尽可能快，因此块的大小只能较小。随着

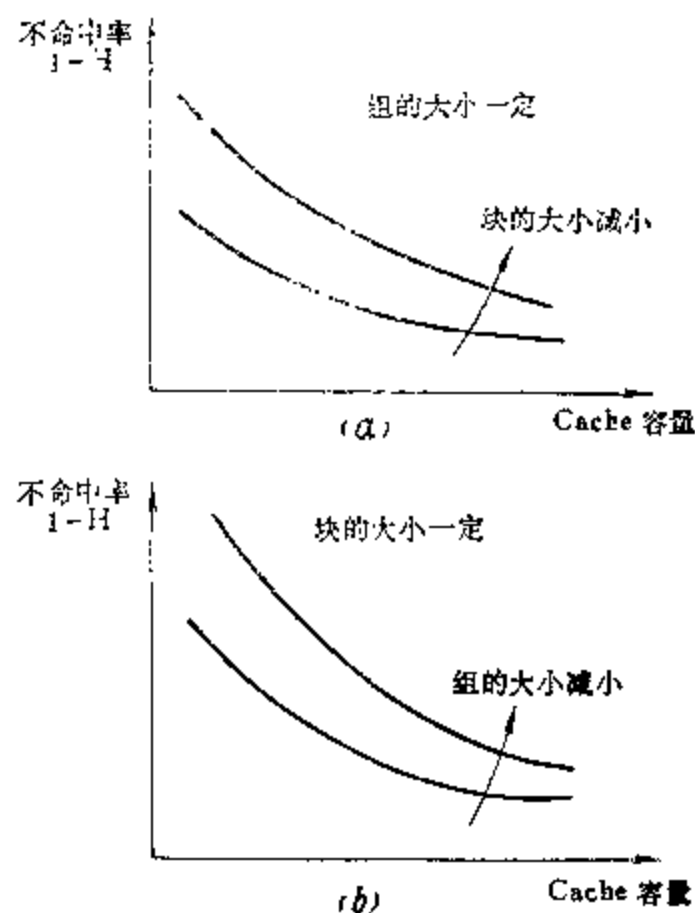


图5.67 块的大小，组的大小与 Cache 容量对 Cache 命中率的影响

高速存贮器片子价格的下降, Cache 的容量不断增大, 已由1K 到8 K, 64K, 不久就会到几百 K 字节。

至于替换算法及题目的不同对命中率的影响, 同虚拟存贮器的情况类似。绝大多数 Cache 采用 LRU 替换算法。

“Cache——主存”层次的命中率应比“主存——辅存”层次的还要高, 这是因为对前者, 调块时处理机是在白等; 而对后者, 调页时处理机可换道。

下面分析“Cache——主存”层次的等效速度与命中率的关系。

设 t_c 为 Cache 的访问时间, t_m 为主存周期, H 为访存地址在 Cache 命中的比例, 则“Cache——主存”层次的等效存贮周期

$$t_a = Ht_c + (1-H)t_m \quad (5.6-1)$$

因此, 采用 Cache 比之于处理机直接访问主存其速度的提高倍数为

$$S_c = \frac{t_m}{t_a} = \frac{t_m}{Ht_c + (1-H)t_m} = \frac{1}{1 - \left(1 - \frac{t_c}{t_m}\right)H} \quad (5.6-2)$$

下面分析对给定的主存和 Cache 速度比值, S_c 最大值与命中率 H 的关系。

令 $H = \frac{k}{k+1}$ 代入 (5.6-2) 式,

则得

$$\begin{aligned} S_c &= \frac{1}{1 - \left(1 - \frac{t_c}{t_m}\right) \frac{k}{k+1}} \\ &= (k+1) \frac{t_m}{t_m + kt_c} \end{aligned} \quad (5.6-3)$$

因为 $\frac{t_m}{t_m + kt_c} < 1$, 因此

$$S_c < k+1$$

即 S_c 可能的最大值恒比 $k+1$ 小。

如果 $H = 0.5$, 相当于 $k=1$, 则不论其 Cache 速度有多高, 其 S_c 的最大值定比 2 小;

如果 $H = 0.75$, 相当于 $k=3$, 则 S_c 的最大值定比 4 小;

如果 $H = 1$, $S_c = S_{cmax} = \frac{t_m}{t_c}$, 这

是 S_c 可能的最大值。

这样, S_c 的期望值与命中率 H 的关系如图 5.68 所示。

由于 Cache 的命中率可比 0.9 大得多, 达 0.996, 因此采用 Cache 是能使 S_c 接近于所期望的 t_m/t_c 。

H 值受 Cache 容量的影响大, 若其典型关系是 Cache 容量为 4 K 字节时, $H = 0.93$; 8K 字节时, $H = 0.97$; 则在 $t_c/t_m = 0.12$, 对于 4 K 字节的 Cache, 其速度提高倍数为

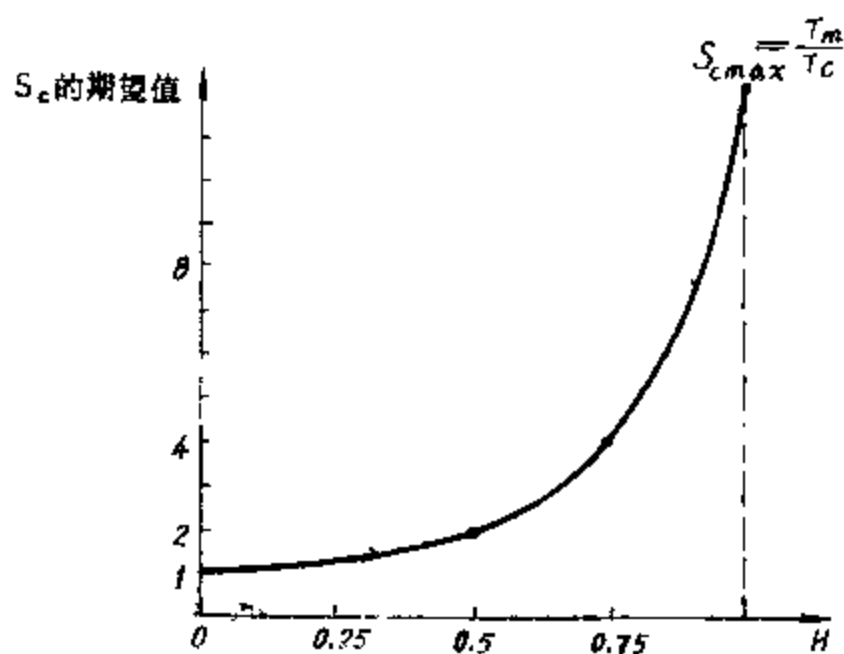


图 5.68 S_c 的期望值与 H 的关系

$$S_{c^{4k}} = \frac{1}{1 - (0.88) \times (0.93)} \approx 5.5$$

而对于 8K 字节的 Cache, 其速度提高倍数为

$$S_{c^{8k}} = \frac{1}{1 - (0.88) \times (0.97)} \approx 6.85$$

因此, 增加 4K 字节的 Cache 容量, 带来层次速度的提高为

$$\frac{S_{c^{8k}} - S_{c^{4k}}}{S_{c^{4k}}} = \frac{6.85 - 5.5}{5.5} \approx 0.24(24\%)$$

显然, 为获得 24% 速度的改进, 这个代价是合算的。

下面再具体看看 Cache 的容量对机器速度的显著影响。对流水机器, 机器速度与主存速度、CPU 拍宽、Cache 容量的可能关系如图 5.69 所示, 机器速度的单位是 MIPS(每秒执行一百万条指令), 主存采用多体交叉存取。

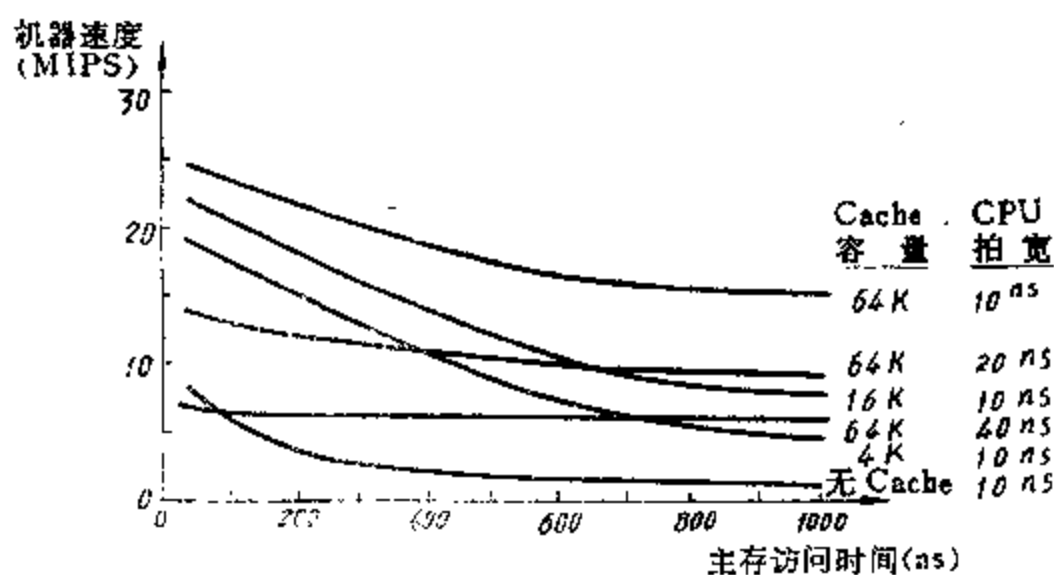


图 5.69 流水机器速度与主存速度、CPU 拍宽和 Cache 容量的可能关系曲线

可见, 主存速度和 CPU 周期一定时, 由不用 Cache 到 Cache 容量由 4K 增大到 64K, 机器速度显著提高, 尤其是在主存速度较低时, 例如, 采用 4K 的 Cache, CPU 拍宽为 10 毫微秒, 主存周期为 1 微秒时, 机器速度约 5MIPS; 同样条件下, Cache 容量增加到 64K, 机器速度可能达 15MIPS, 而如果没有 Cache 时, 机器速度可能只有 2MIPS。

还可看出, Cache 容量的增大, 可以显著降低对主存速度的要求。例如要达到机器速度为 15MIPS, 对于 10 毫微秒 CPU 拍宽, 4K 容量的 Cache, 要求主存访问周期为 200 毫微秒, 而当 Cache 容量增大到 64K 时, 主存周期可以降低到 1 微秒。如果不采用 Cache 技术, 存贮系统的速度是很难与 CPU 的速度相适应; 而且当 Cache 容量足够大后, 主存速度对机器速度的影响就会显著减小。

“Cache—主存”层次就讨论到此。至于“Cache—主存—辅存”三级层次, 它是“主存—辅存”层次和“Cache—主存”层次的组合。对于一个虚地址, 若对应的单元已在 Cache, 则需访问 Cache, 为此就需把虚地址变换成 Cache 地址; 若对应的单元已在主存, 但还没调入 Cache, 则需访问主存, 为此就需把虚地址变换成主存地址, 对于读访问, 还一定得把这个单元所在的块调入 Cache; 若对应的单元还不主存, 就得由辅存调页进主存, 为此就需把虚地址变换成辅存实地址和主存地址。对于头一种情况, 当然宜于直接由虚地址

变换成 Cache 地址，而不是先由虚地址变换成主存地址，再由主存地址变换到 Cache 地址，因为后面的方法显然会增长地址变换的时间。不过，虚地址到 Cache 地址的变换机构和虚地址到主存地址的变换机构有些部分可以共用。当然，构成多级层次还有很多其它问题，但只要掌握了前述“主存—辅存”和“Cache—主存”这二级层次的基本概念和分析方法，那都是可以解决的。

存贮体系的设计不能只从存贮体系本身出发，而必须联系处理机的要求和操作系统的需要全面考虑。

§ 7 主存保护与主存控制部件

7.1 主存保护

近代计算机系统要设计成其资源能被共同执行的多个用户所共用。就主存来说，它就同时存有多用户的程序和系统软件。为使系统能正常工作，应防止由于一个用户程序出错而破坏同时存在主存的系统软件或其它用户的程序，还要防止一个用户程序不合法地访问不是分配给它的主存区域，那怕这种访问不会引起破坏。为此，系统结构需为主存的使用提供存贮保护，它是多道程序和多处理系统所必不可缺的。

首先讲存贮区域的保护，然后再来讲访问方式的保护。为实现区域保护，对于不是虚拟存贮器的主存系统可采用第二章讲过的界限寄存器方式，由系统软件经特权指令置定上、下界寄存器从而划定每个用户程序的区域，禁止它越界访问。由于用户程序不能改变上、下界的值，因此不论它如何出错，也只能破坏该用户自身的程序，侵犯不到别的用户程序及系统软件。

然而，界限寄存器方式只适用于每个用户程序占用主存一个或几个（当有多对上、下界寄存器时）连续的区域；而对于虚拟存贮器系统，由于一个用户的各页能离散地分布于主存内，从而无法使用这种保护方式。对虚拟存贮器的主存区域保护就需采用页表保护和键式保护等方式。

先讲页表保护。前面 § 2 和 § 5 都讲过，每个程序有它自己的页表，它的大小（行数）等于该程序的虚页数。若它有四页，则只能有 0、1、2、3 四个虚页号，设由操作系统建立的该程序页表如下：

虚 页 号	实 页 号
0	4
1	8
2	10
3	14

那么不论该程序的虚地址如何出错，它也只能影响到分配给该段的主存 4、8、10、14 号实页。假设虚页号错成“5”，它必然不可能在该程序的页表中找到，从而访问不了主存，也

就侵犯不了主存中其它程序的区域。这正是虚拟存贮器系统本身固有的保护机能，也是它的一大优点。为了更便于实现这种保护，还可在段表(前面讲过，可把它看成是页表层次中的第一级，不只是段、页式管理中才有)中的每行内，不仅设置页表起点，还设置段长(页数)项。若出现该段内的虚页号大于段长，则可发越界中断。

这种页表保护是在没形成主存实地址前进行的保护，使之无法形成能侵犯别的程序区域的主存地址。然而，若地址形成环节由于软、硬方面的故障而形成了不属于本程序区域的错误主存地址时，则上述这种保护就无能为力了。因此，还应采取进一步的保护措施，键方式是其中成功的一种。

键方式是由操作系统，按当时主存的使用分配状况，给主存的每页配一个键，称为存贮键，它相当于一把“锁”。所有页的存贮键存在主存相应的快速寄存器内，每个用户(任务)的各实页的存贮键都相同。为了打开这个“锁”，需要有把“钥匙”，称为访问键。每个用户的访问键由操作系统给定，存在处理机的程序状态字(PSW)或控制寄存器中。程序每次访问主存前，要核对主存地址所在页的存贮键是否与该程序的访问键相符，只有相符，才准访问。这样，就是错误地形成了侵犯别的程序的主存地址，也因有这种键保护而仍然不允许访问。IBM370的保护键有4位，能表示已调入主存的16个活跃的程序。其中“0000”访问键是操作系统的，对这个访问键不论是否和存贮键相符都可访问，这是操作系统应能访问到主存整个区域所要求的。

上面讲的保护是保护别的程序区域不被侵犯，但不是保护正在执行的程序本身不被破坏。如何实现对正在执行的程序的重要关键部分的保护也是存贮保护的一部分，可以有各种办法，环状保护是其中的一种。

这种保护把系统程序和用户程序按其重要性及对整个系统能否正常工作的影响程度分层，如图5.70所示。设0、1、2三层是系统程序的，之外的各层是同一用户程序的分层。环号大小表示保护的级别，环号愈大，等级越低。在现行程序运行前，先由操作系统定好程序各页的环号，并置入页表，而后把该道程序的开始环号送入处理机内的现行环号寄存器，并且把操作系统规定给该程序的上限环号(规定该程序所能进入的最内层环号)也置入相应的寄存器。

若 P_i 是在某一个时候属 i 层各页的集合，则当进程执行 $P \in P_i$ 页内的程序时，允许访问 $q \in P_j$ 页，这里对应的是 $j \geq i$ 。但是如果是 $j < i$ 时，则需由操作系统环控制例行程序判定这个向内访问是否允许、是否正确之后才能访问，否则就是出错，进入保护处理。但 j 值肯定不能小于给定的上限环号。只要 $j \neq i$ ，就进入中断，若允许访问，则需经特权指令把现行环号寄存器的值由 i 改为 j 。

这种环式保护既能保护不要由于用户程序的出错而侵犯系统程序，也能保护不要由于同一用户程序内的低级(环号大)的部分的出错而破坏其高级(环号小)的部分。这种环式保

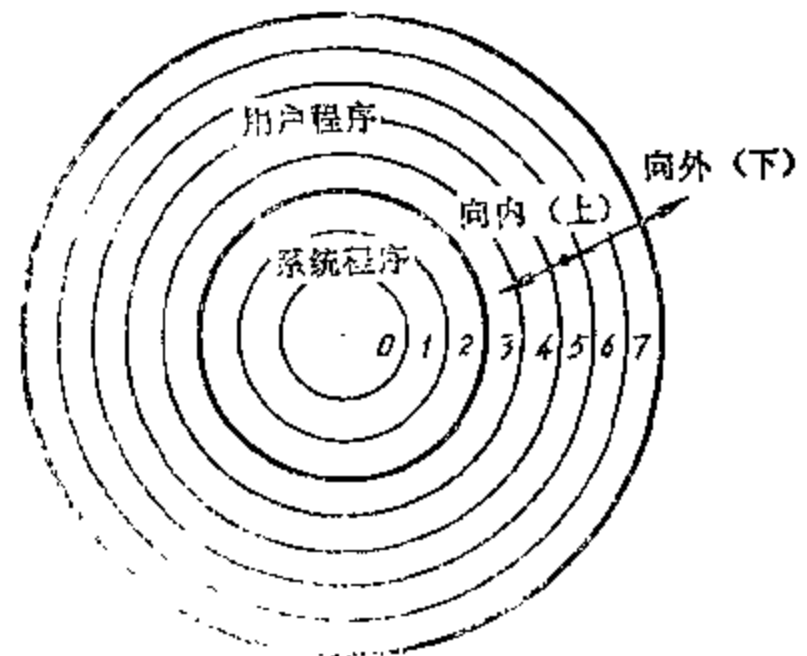


图 5.70 环式保护的分层

护对系统程序的研究和试运行特别有利，因为可以做到能修改系统程序的某些部分而不必担心会影响到系统程序已设计好并调好的核心部分。

至于如何控制 $j \neq i$ 的跨层访问，有的机器规定只能由规定的转移指令执行，且对 $j > i$ 和 $j < i$ 分别只能用不同的转移指令。

上述种种区域保护，如判越界、判键相符、判环号相符、判不超出段长等等，当然都是经硬件实现，因此速度可以很快。

上面讲的区域保护是对允许访问的区域可以进行任何形式的访问；而对允许区域之外，则任何形式的访问都不允许。但在实际中，只是这种限制往往适应不了各种应用的要求，因此还得加上访问方式的限制（保护）。

对主存信息的使用可以有读（R）、写（W）和执行（E）三种方式，“执行”指的是按指令来用。相应的也有 R、W 和 E 访问方式保护。这三者的逻辑组合可以反映出各种应用要求。如：

$\overline{R} \vee \overline{W} \vee \overline{E}$ ——不允许进行任何访问（如专用的系统表格）；

$R \vee W \vee E$ ——可以进行任何访问；

$R \wedge \overline{W} \vee \overline{E}$ ——只读访问（如对各个用户都用到的表格常数）；

$(R \vee W) \wedge \overline{E}$ ——只能按数据进行读、写（例如阵列数据当然不能按指令执行）；

$\overline{R} \vee \overline{W} \wedge E$ ——只能执行，不能按数据使用（如某个专门的程序）；

$\overline{R} \vee \overline{E} \wedge W$ ——只写访问（如用户对操作系统缓冲器的写入）；

$(R \vee E) \wedge \overline{W}$ ——不准写访问；

等等。

对前面讲过的各种区域保护，都可加上相应的访问方式位以实现这种访问限制。

例如，对界限寄存器方式，可以在下界寄存器加一位访问方式位。它为“0”时，表明该区域可读、可写；它为“1”时，表明只能读、不准写。DJS-200 系列就是这样用的。

又如，对键方式，也可以加上访问方式位。有时，它的作用还需和访问键与存贮键是否相符一起考虑。IBM370 就设计成当“读”保护位为“0”时，不论是否键相符，恒可进行读访问；当“读”保护位为“1”时，只有键相符时才能进行读访问；但对写访问均只有键相符时才能进行。这里，对于写访问因保护而被禁止时，被保护页的内容保持不变；对于读访问因保护而被禁止时，被保护页的内容既不能取到寄存器，也不传送到其它页或 I/O 设备。显然，对 IBM370 通过使读保护位为“0”，可实现主存只准读的部分的共享。

至于环式保护和页表保护，可以把 R、W、E 等访问方式位设在各个程序的段、页表的各行内，使得同一环内或一段内的各页可以有上述种种不同的访问保护，以增强灵活性。而且还可做到各个用户对同一共享页的访问方式不同。例如有的用户只能读访问，而有的用户则是读、写访问都允许。对于环式保护还可做到使访问保护不仅取决于 R、W、E 的组合，还要取决于 j 比 i 是大、是小、还是相等。

上述按访问方式位的组合以及与键相符、环号比较等配合来决定允许进行何种访问也是经硬件实现。

除了以上这些保护外，在某些应用中，我们既要求能实现多个用户可读、写访问共享的数据，但又要保证只当一个用户访问完该数据后，别的用户才可访问，以防止在一个用户还

未把某个共享文件写好之前，别的用户却能把它读了去。利用第二章讲过的 IBM370 的“测试与置定”和“比较与交换”指令能实现这点。

7.2 主存控制部件

早期的机器是直接指令中的地址码去访问主存的存贮体，而且读出的信息不必经过任何变换直接送往运算器。但是近代的机器，指令中的地址码只是虚地址，它需要经过复杂的变换才能变成访主存的真（实）地址；而且，在真正访问存贮体之前还得经过存贮保护的检查。此外，由存贮体读出的信息是不能直接送往处理机的，它需要经过错误检测以及错误纠正。还有，处理机的编址方式不只是早期机器的按字编址，而且还有按字节编址直至按位编址；采用多体并行交叉，单体多字等结构，这就需把存贮体读出的信息经过合并或分离之后才能成为处理机所需的信息。若在主存和处理机之间加了 Cache 存贮器，就还得控制虚地址是访问主存，还是访问 Cache。由于机器结构从以 CPU 为中心发展到以主存为中心，因此还得协调好多个 I/O 处理机（通道）和 CPU 对主存的访问；对于多处理机系统来说，与主存相连的处理机数目更多，从而这种协调、控制就更复杂。以上种种就使得主存系统除存贮体外，还得要有复杂的主存控制部件（存控）来完成这些功能。不同的机器对存控的安排不同，有的是把上述的某些功能，如主存地址的形成和读出信息的合并或分离放在处理机去实现。然而，不论放在哪里，上述这些控制功能总是必须有的。

关于虚、实地址变换，存贮保护，多体并行交叉，单体多字和 Cache 等我们在前面都已讲过了。而错误检测和纠正以及提高主存可靠性、可用性、可维护性的某些措施在下一章再讲。

当多个 I/O 处理机和多个 CPU 都对主存提出访问申请时，存控应当有排队控制逻辑，安排好这些访问的先后次序，以尽量提高访问效率，减少访存冲突。对于单 CPU 系统，访主存申请有 Cache 访存申请、通道访存申请，CPU 直接对主存的写数申请、CPU 直接对主存的读数申请、CPU 取指令访存申请等等。一般来说，这些申请的优先次序就是上述的先后次序，即 Cache 访存申请级别最高，取指令访存申请级别最低。Cache 的申请级别之所以最高，前已说过，是因为 Cache 调块时，CPU 在白等。至于通道的申请级别比 CPU 高，主要原因是通道的缓冲量小，如不及时响应其访存申请，就有可能造成 I/O 信息的丢失；而且通道的申请中包括取通道控制字和取通道地址字，如不及时提供就会造成通道和外设的不正常或错误动作。而对 CPU 的写数、读数和取指令这三种申请，在重迭或流水的机器中，由于会有第三章讲过的地址相关问题，所以要把写数申请放在读数申请之前，使得读出来的确是在它之前应写好的数；取指令申请级别之所以排在最低是因为迟些取得指令不会造成整个机器工作的错误，而且对有指令缓冲器的机器，还可与指令的执行重迭，插在主存有空闲时去取。

大多数机器是把访存级别定死的，但也有些机器在程序的执行过程中还可动态地改变。有的机器的主存系统只有一个入、出端，由存控的排队线路安排好各个申请的响应次序；有的机器则有多个入、出端，各个端的优先级别不同，对每个端的申请可再进一步分级。

随着系统结构的发展，还可能把更多的逻辑功能分散到存控来。例如，第二章讲过的，对于“测试与置定”和“比较与交换”等指令，就得要求是由存控来保证对指定单元的读和写之间不能被别的指令和处理机所访问。

在上述对存控的所有要求都确定之后,就可进行对存控的具体逻辑设计,在设计时一定要努力减少由虚地址到存贮体地址之间和由存贮体输出到处理机相应寄存器之间的级数,尽可能不要由于上述种种要求而使访存时间过分增加。

总之,存控是改善系统的吞吐能力,使之高速可靠工作的不可缺少的部件,也是使存贮体系得以有效工作的重要组成部分。

主要参考文献

- [1] R. E. Matick, "Computer Storage Systems and Technology," John Wiley & Sons, 1977.
- [2] J. P. Hayes, "Computer Architecture and Organization," McGraw-Hill, 1978.
- [3] D. J. Kuck, "The Structure of Computers and Computations," Vol. 1, John Wiley & Sons, 1978.
- [4] A. S. Tanenbaum, "Structured Computer Organization," Prentice-Hall, 1976.
- [5] D. J. Kuck (ed.), "High Speed Computer and Algorithm Organization," Academic Press, 1977.
- [6] C.G. Bell, "Computer Engineering—A DEC View of Hardware Systems Design," Digital Press, 1978.
- [7] R. L. Sites, "Operating Systems and Computer Architecture," Ch. 12 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 1980.
- [8] A. C. Shaw, "The Logical Design of Operating Systems," Prentice-Hall, 1974.
- [9] S. E. Madnick and J. J. Donovan, "Operating Systems," Mc Graw-Hill, 1974.
- [10] IBM 4300 Processors Principles of Operation for ECPS, VSE Mode, GA22-7070-0, 1979.
- [11] P. J Denning, "Virtual Memory," Computing Surveys, Vol. 2, No. 3, Sept. 1970, pp. 153—189.
- [12] S. L. Rege, "Cost, Performance, and Size Tradeoffs for Different Levels in a Memory Hierachy," Computer, Vol. 9, No. 4, April 1976, pp. 43—53.
- [13] Makoto Terajima, "Recent Progress in Memory Devices and their Prospects," IFIP 80, Oct. 6—9, 1980, pp. 127—135.
- [14] Y. S. Lin and R. L. Mattson, "Cost-Performance Evaluation of Memory Hierarchies," IEEE Trans. on Magnetics, Vol. 8, No. 3, sept. 1972, pp.390—392.
- [15] J. Bell, et al, "The Investigation of Alternative Cache Organizations" IEEE Trans. on Computers, Vol. 23, No. 4, April 1974, pp. 346—351.
- [16] W. D. Strecker, "Cache Memories for PDP—11 Family Computers," The 3th Annual Symposium on Computer Architecture, 1976, pp. 155—158.
- [17] G. J. Burnett and E. G. Coffman, "A Study of Interleaved Memory Systems," AFIPS Conf. Proc., Vol. 36, 1970, pp. 467—474.

第六章 可靠性技术

§ 1 基本概念

随着计算机应用领域的日益扩大以及实时处理和实时控制系统的发展,对计算机的可靠性要求越来越高。例如在电话交换系统、航空和航天控制系统、铁路交通控制系统中,那怕是一个细微的故障,也可能会带来灾难性的后果。所谓可靠性就是指在给定的时间内计算机系统能正常运转的概率。通常可靠性用平均无故障时间(MTBF, Mean Time Before Failure)来表示。所谓平均无故障时间是指系统能正常工作的时间的平均值。显然,此时间越长,就表明计算机系统可靠性越高。提高计算机系统的可靠性一般有两条途径。一条是提高元器件的质量,严格地老化筛选,改进工艺结构和完善逻辑设计。这是一条主要的途径。但是,组成计算机的元件不可能是绝对可靠的。不论怎样提高元器件的质量,系统出故障的可能性总是存在的。因此提高可靠性的另一条途径是发展容错技术。所谓容错,就是容许故障发生。因此,容错技术可定义为:“尽管硬件有故障或程序有错误,系统仍能正确执行特定算法的能力”。

这里必须指出,可靠性指标并不是唯一指标。当计算机出故障而失效后,必须进行维修。在维修期间,计算机一般不能被正常使用。这就降低了计算机的使用效率。因此,可维修性就成了另一个指标。可维修性是指计算机的维修效率,通常用平均修复时间(MTRF, Mean Time Repair a Fault)来表示。所谓平均修复时间是指从故障发生到故障修复平均所需要的时间。显然,平均修复时间越少,可维修性就越高。另外,可用性也是另一重要指标。可用性就是指计算机的使用效率。它是以系统在执行任务的任意时刻能正常工作的概率来表示。因此,它可以表示为

$$A = \frac{MTBF}{MTBF + MTRF}$$

综上所述,衡量一个计算机应该是可靠性R、可用性A和可维修性S三者的综合。当然不同用途的计算机对这三个指标可有不同的侧重。

一般,所谓的RAS技术,就是可靠性、可用性和可维修性技术的总称。

在介绍容错技术之前,先讲一下计算机常见故障的类别及产生的原因。

1.1 故障的类别及产生的原因

计算机系统在交付使用后,随时都有可能出故障。硬件故障是导致一个系统、部件、元件不能按设计要求工作的物理原因,象短路,断路、虚焊等。显然,要提高计算机的可靠性,对计算机系统内的各种故障进行分类、分析及模型化是首要的一步。

从不同的角度出发,可以对故障进行不同的分类。

首先,按计算机系统内的界面分,故障可分为软件故障和硬件故障两大类。软件故障是指为系统运行所编制的程序(包括系统程序和用户程序)本身的错误。这些软件故障可能是在程序的指定阶段和设计阶段,也可能是在指令化阶段被引入的。从本质上讲,这是一种设计性故障。硬件故障是指在元器件中发生的故障。它又可以分为逻辑故障、参数故障等。凡是使电路中某单元的逻辑函数值取与原来值相反的值的故障,都归为逻辑故障。其中以“固定1”(S—a—1)和“固定0”(S—a—0)形式出现的逻辑故障最为普遍。“固定1”(或“固定0”)是指在电路某点出现了永久的“1”(或“0”)信号。各种断路、短路、碰线,直至元件损坏都可以归结为逻辑故障。如图6.1所示,如果门 G_1 的输出端与正电源短路,在线5,线6,线7处都将发生“固定1”的故障;如果在 G_2 的6输入端发生断路,它将等效于线6处的“固定1”故障,而线5,线7处没有故障。如果说线6的“固定1”故障是逻辑故障的话,那么 G_2 的6输入断路应称作为物理故障。把某些物理故障等效为(或模型化为)某些逻辑故障,把对故障的分析变成一个逻辑问题,是故障测试与诊断中的一个重要思路。参数故障是指元器件的某些参数的变化超出了允许的范围。如时钟频率增高、电源电压降低、掉电等。

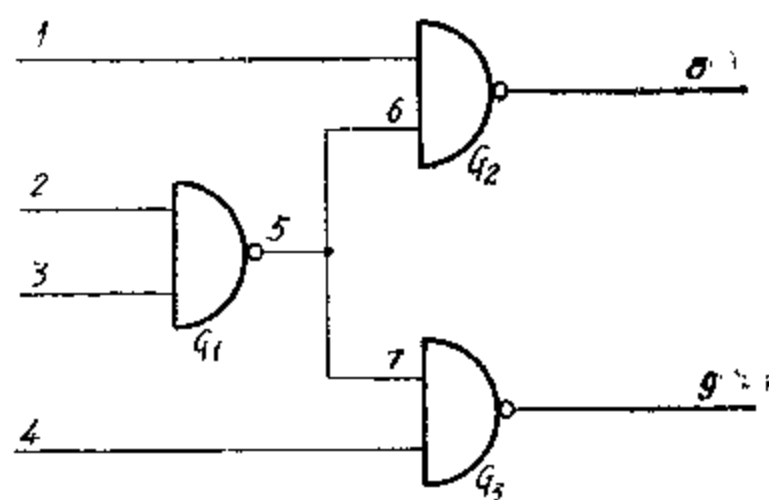


图 6.1 逻辑故障例

如果按故障持续时间分类,故障可分为固定性故障和暂时性故障。固定性故障是由电路元件的变质,电路内部的短路或断线,软件设计不周等原因引起的必然性故障。它的特点是故障现象可以重复出现,在无校正设备的情况下,只有人工干预,系统才能恢复正常。因此,对于固定性故障,可以通过诊断程序来进行故障定位。暂时性故障是由外界电网电压的跳动、外部电磁性干扰信号或某连接处的虚焊等原因所引起的偶然性故障。它的特点是持续时间短,往往不要人工干预,系统又能自动恢复正常工作。因此,暂时性故障就很难进行诊断定位。但另一方面,正因为故障是瞬时性的,系统可以通过下面将要说的指令复执或程序卷回重新执行被认为是错误地执行了的指令或程序,以清除故障所引起的错误。

看出,故障产生的原因也是多种多样的。有的故障是由设计错误引起的,有的故障是由元件的变质以及外界干扰引起的。设计错误包括程序设计错误和硬件逻辑设计错误。为了减少程序设计错误,程序在交付使用前应进行比较彻底的测试和调试,或者用某种方法证明其正确性。为了减少硬件逻辑设计错误,一般是采用模拟的办法,在硬件实现之前,用模拟程序验证该硬件的逻辑设计的正确性(包括各种时序关系的正确性)。如果在模拟阶段发现有逻辑设计错,就可以及时纠正,从而大大提高了将要实现的系统的可靠性。设计性故障的主要特点是:一旦排除了,就不再重现。但由元件变质、失效引起的故障就不是如此。任何元件都不能无限期使用。长时期的使用、生产时带来的缺陷以及电源的一些瞬时不良行为都可能引起元件的变质,甚至完全失效。外界干扰一般指电源波动、电磁干扰、环境温度和机械振动等等。由这些干扰引起的故障是暂时性故障,是没有办法预先排除的,因此要求计算机能有容错能力。

1.2 冗余技术

要使计算机具有容错能力，基本的方法是采用冗余技术。冗余就是“多余”的意思。我们给计算机系统增添一些硬件设备，这些设备从实现系统功能的角度来讲是“多余”的。没有这些设备，系统照样能完成预定的功能，而增添这些设备的目的仅在于提高整机的可靠性，这叫硬件冗余。同样，软件冗余是指给系统增添的程序。另外，在有容错能力的计算机发现系统出错后，经过一定的恢复过程，系统有可能恢复正常工作。但刚才还没执行完的指令，应被重新执行。这叫指令复执。如果最近被执行的一段程序被认为是在错误的系统环境下运行的，那么这一段程序应被重新执行，以消除错误执行的程序对系统的影响，这叫程序卷回。程序重新开始执行的入口称为卷回点。显然，要使程序能真正实现卷回运行，在每一个卷回点，系统都应保护有关的信息。指令复执和程序卷回都属于时间冗余。时间冗余就是以牺牲一定的时间为代价，来求得系统的容错能力。还有一种冗余叫信息冗余。它是指系统内增添的某些数据，这些冗余数据是为提高可靠性所需要的。被广泛采用的纠错码就是信息冗余的一个典型例子。设系统内信息存贮和传输的单元是 k 位二进制信息，如果另外再添加 r 位校验位，构成 $n = k + r$ 位的二进制信息。增添的 r 位信息，虽然对原 k 位信息本身来讲是多余的，但它却使系统对信息位具有某种检错或纠错的能力，这 $n = k + r$ 位的二进制信息就叫纠错码。关于纠错码，在 §3 详细讨论。

可以说，上述四种冗余技术是实现容错，提高整机可靠性的基本手段。

这里有必要对硬件冗余和软件冗余作进一步说明。

硬件冗余可分为静态冗余和动态冗余两种。静态冗余（也称掩蔽冗余）是指采用串并配置和三模冗余（TMR, Triple Modular Redundancy）等逻辑重迭技术来有效地“掩蔽”硬件故障。如图 6.2 中的三模冗余所示，三个相同的模块都从输入端接受输入信息，三个模

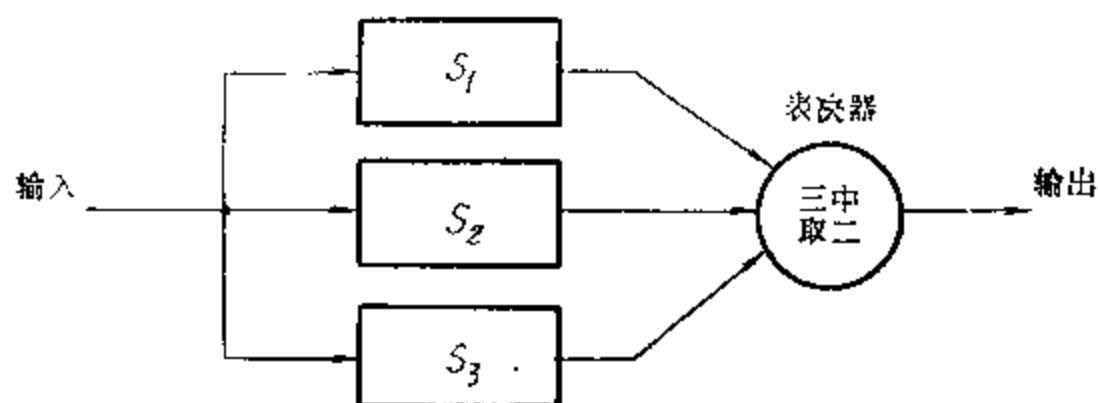


图 6.2 三模冗余

块同时工作，得出的三个结果送入表决器进行多数表决。表决器以多数结果作为最后输出信息。也就是说，如果三个模块中有一个失效，最后输出仍是正确的结果。从外部看，我们根本不知道哪一个模块出故障，甚至根本就不知道有模块出故障。这实际上是二个正确的模块把失效的模块的错误掩盖起来了，这就是“掩蔽”的意思。显然，对静态冗余来讲，在部件出故障后，系统无须进行诊断（诊断就是发现计算机的故障并确定故障位置的过程），也无须进行修复（修复就是修理有故障的计算机系统，使之恢复到正常状态），就能实现容错运

行。这是静态冗余的最重要的优点。特别是在对付暂时性故障时，更显示出它的威力。因为暂时性故障持续时间很短，真正要进行诊断定位时，故障又不存在了。事实上，对暂时性故障也没有进行诊断和修复的必要。静态冗余的代价比较昂贵。动态冗余（也称选择冗余）是采用辅助子系统来对主系统进行检测和故障定位。一旦检测到故障，并诊断出故障发生的位置后，系统进行自动修复。因此动态冗余应包括三步：故障检测、故障定位和系统恢复。如果检测是和用户程序的运行同时进行的，就称为并行检测。象利用纠错码进行检测等都属于这一类。另一种叫周期性检测。系统内有专门的监督器定期地中断用户程序，利用辅助子系统进行检测。一旦发现故障后，立即进入第二步，进行故障定位。衡量这二步工作好坏的标准是故障复盖率、诊断时间和分辨率。故障复盖率是能检测出来的故障的数目与故障总数目之比。分辨率是指故障定位细化的程度。故障定位越细，分辨率就越高。显然，应尽量提高故障复盖率、缩短诊断时间和提高分辨率。在故障定位之后，有二种恢复手段。系统可以在操作系统的配合下自动置换故障部件（称为自修理），也可以自动重新配置系统，以组成一个不同的计算机系统（称为降级）。新组成的系统在功能上可能不如原系统强，但在一定的范围内仍能执行一定的任务。例如，图 6.3(a)表示的是一个多处理机系统。它共有 n 个处理机 ($P_1 \cdots P_n$) 和 m 个存贮器 ($M_1 \cdots M_m$)，通过联接网络和输入设备 I 和输出设备 O 组成了一个系统。假定在原系统中每一个处理机可以对任何一个存贮器进行存取。如果通过诊断发现 M_1 有故障，则系统可以自动重新配置，使得各处理机只可以对 $M_2, M_3, \cdots M_m$ 进行存取，这样就组成了图 6.3(b) 的系统。这个系统的主存容量比以前小，因而在功能上有所减弱，但仍可在容许的范围内运行。所谓降级，就是在容许的范围内降级使用。同样，在诊断出 M_1 有故障后，也可以通过切换装置自动用备用的存贮器 M' 来代替 M_1 。这就成了图 6.3(c) 的系统。无论是降级，还是自修理都可以由切换开关来实现，也可以用软件来实现，也可以是二者的结合。接着，系统必须对系统内关键性信息所受到的破坏作出估计，重建这些信息。最后通过程序卷回，重新执行某些任务。动态冗余所化的代价比较小，硬设备的利用率也较高。这是动态冗余的优点。但用户程序要被诊断过程所中断，系统恢复过程较慢，因此系统在时间上所化的代价却较大。再加上它比较难以消除暂时性故障的影响，所以在一些实时系统中，静态冗余仍采用得相当普遍。在一般情况下，是二者的结合。如上面提到的用来进行故障检测和故障定位的辅助子系统，其本身应该是十分可靠的。这一部分硬件一般称为硬核。硬核就是系统中可被认为是无故障的部分（可靠性接近于 1）。而硬核往往是用静态冗余进行严格保护的。一般硬核只占整机的很小一部分，对这一部分采用静态冗余对整个系统的造价影响也不大。

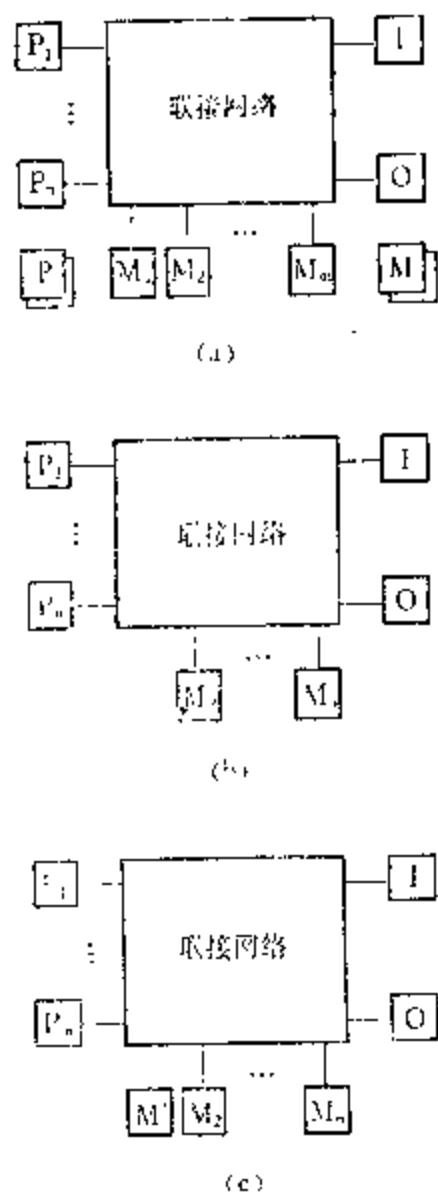


图 6.3 多处理机系统的降级

称它为用软件实现的硬件冗余。另一类软件冗余是为了对付软件故障而引入的冗余程序。用户程序，不管在交付使用前如何进行过彻底的测试和调试，仍可能含有软件故障。为了使系统在软件有故障的情况下仍能运行，对同一个任务可以多编写一些用不同的算法完成同一任务的程序。这些冗余的程序和原本的程序，再加上接受测试程序(AT, Acceptance Test)，就构成了一个恢复块。如图 6.4(a)所示。图中双竖线之间括起来的是恢复块，一条竖线之间括起来的是原本程序和冗余程序。控制进入恢复块 A 后，先运行原本程序 AP，完毕后进行接受测试 AT。接受测试往往是根据预先给定的程序结果的所在范围，来检查此程序得出的结果是否正确。如果结果不正确，或者程序 AP 根本就没有能执行到底（如出现了企图以零作除数的程序性中断情况等），就转向执行程序AQ。当然，在 AQ 执行之前，系统应恢复到刚进入恢复块A时的状态。原本程序和冗余程序中只要有一个通过了接受测试，恢复块的任务就算完成，进而执行恢复块以后的程序。如果这些程序都没有能通过接受测试，恢复块 A 就宣告失效。恢复块可以有嵌套结构。见图 6.4(b)，这里整个是一个恢复块 A。它由原本程序 AP、冗余程序 AQ 和接受测试 AT 组成。而 AP 又包含一个恢复块 B，AQ 又包括二个恢复块 C 和 D。现在假定 BP、BQ、BR 均没有能通过接受测试 BT，则恢复块B 失效。这导致了程序 AP 的中止，系统状态再进一步恢复到刚进入恢复块 A时的状态，转而执行冗余程序 AQ。显然这种软件冗余与用软件实现的硬件冗余是完全不同的二个概念。

综上所述，计算机可靠性技术应包括：

(1) 系统的可靠设计：近年来发展起来的软件工程学的一个任务就是研究软件的可靠设计。至于硬件的可靠设计与计算机设计自动化直接有关；

(2) 故障诊断；

(3) 纠错码的应用；

(4) 可靠性模型和分析；

(5) 容错技术的实现。

在本章的以后各节中我们将对后四个部分分别进行讨论。

```

A: ensure AT
    by AP: begin
                <program text>
            end
    else by AQ: begin
                <program text>
            end
    else error
  
```

(a) 恢复块

图 6.4 软件冗余例

A; ensure AT

```
by AP; begin declare y
    <program text>
    B; ensure BT
        by BP; begin declare U
            <program text>
            end
        else by BQ; begin declare V
            <program text>
            end
        else by BR; begin declare W
            <program text>
            end
        else error
    <program text>
    end
else by AQ; begin declare Z
    <program text>
    C; ensure CT
        by CP; begin
            <program text>
            end
        else by CQ; begin
            <program text>
            end
        else error
    D; ensure DT
        by DP; begin
            <program text>
            end
        else error
    end
else error
```

(b) 有嵌套的恢复块

图 6.4 软件冗余例

§ 2 故障诊断

故障诊断包括两个方面,即故障检测和故障定位。测试并确定计算机有无故障的过程称为**故障检测**;判定故障发生在某个子系统、功能块、组件或元件的过程称为**故障定位**。无疑这二者对动态冗余是必要的。即使在静态冗余系统中,任何元件最后总要损坏。因此,及时检测和定位出故障元件以便能及时更换它,才能使静态冗余系统保持原有的容错能力。另外,故障诊断对系统调试和维护也是十分重要的。如果一个计算机系统的诊断能力很低,不仅对维护人员是一个负担,而且也必然会大大降低系统的使用效率。故障诊断的方法很多。早期常用机器语言编写的“检查诊断程序”来检查系统并查出故障,但它有局限性。近年来在故障诊断方面提出了一些新方法,例如故障定位测试法(Fault Location Test)和针对微程序设计计算机的微诊断法。

2.1 故障定位测试法

故障定位测试法是一种系统自动诊断的方法。它直接以电路作为诊断对象,将被测试的系统划分成许多测试区,并向这些区域发送一系列测试码,然后回收并分析被测试区域的响应,以找出故障位置或找出产生故障的元件。假如图 6.5 所示的电路是被测试的区域,并假定在 G_0 的输出端线 11 处有“固定 0”的逻辑故障。因为只有输入端 X_1, X_2, \dots, X_7 和输出端 Z 是可被外界接触到的,我们要检测出此电路有“固定 0”故障,且把故障定位到线 11 处,只有在输入端上加上一定的输入向量 $X = (x_1, x_2, \dots, x_7)$,使得输出端 Z 的值在有此故障和无此故障时正好相反。为了让 G_0 的线 11 处的“固定 0”故障显示出来,首先应让线 11 处在正常情况下的逻辑值是“1”。如这条件成立,我们就说线 11 处有出错信号 D 。出错信号有两种: D 表示正常情况下为“1”,故障情况下为“0”; \bar{D} 表示正常情况下为“0”,故障情况下为“1”。为了让出错信号传到 Z 端被外界感知,线 10 必须为“0”,线 12 必须为“1”。这样,线 13 会出现 D ,线 14 会出现 \bar{D} ,线 15 即输出端 Z 会出现 D 。反过来,由

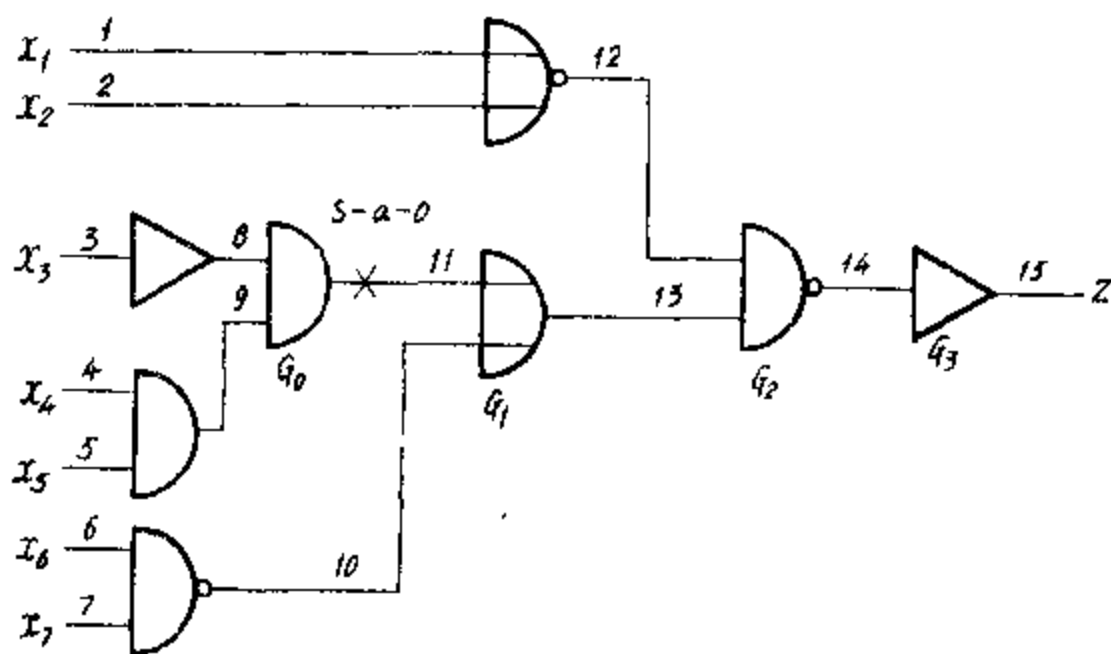


图 6.5 故障定位测试例

于线12为“1”，线10为“0”，所以，应有 $x_1=0$ ， $x_2=0$ ， $x_6=1$ ， $x_7=1$ 。又因为 G_0 正常输出应为“1”，所以线8和线9均应为“1”，由此得 $x_3=0$ ， $x_4=1$ ， $x_5=1$ 。这样，输入向量 $X=(x_1, x_2, \dots, x_7)$ 应为 $X=(0,0,0,1,1,1,1)$ 。如果此 X 加到输入端上，无故障时， $Z=1$ ；线11处有“固定0”故障时， $Z=0$ 。也就是说， G_0 的“固定0”故障沿着通路 $G_0-G_1-G_2-G_3$ 传播到 Z 端。这条通路称为敏化通路。应该看到，此输入向量 X 除了能检测出线11处的“固定0”故障外，还能检测出敏化通路上的其它故障，即线13的“固定0”故障、线14的“固定1”故障和线15的“固定0”故障。我们把这一输入向量称为这些故障的测试模式，简称为测试。当然仅用这一测试是不能把这四个故障区分开来的。也就是说，只用这一测试还不能把故障定位到某一根线上。如果一个测试能使电路对二个故障输出不同的值，就称这个测试区分了这二个故障。如果我们把能检测出测试区域内所有的故障，且能区分出任何二个故障的测试及对应的响应汇总起来，就可以构成这个测试区域的故障字典。根据此故障字典可以形成一定的测试程序。

一般是用另一台计算机控制的自动测试装置进行诊断的。自动测试装置的构造见图6.6。它先根据测试程序的要求，产生一定的输入 X ，输入到被测单元(UUT)中去，UUT的响应

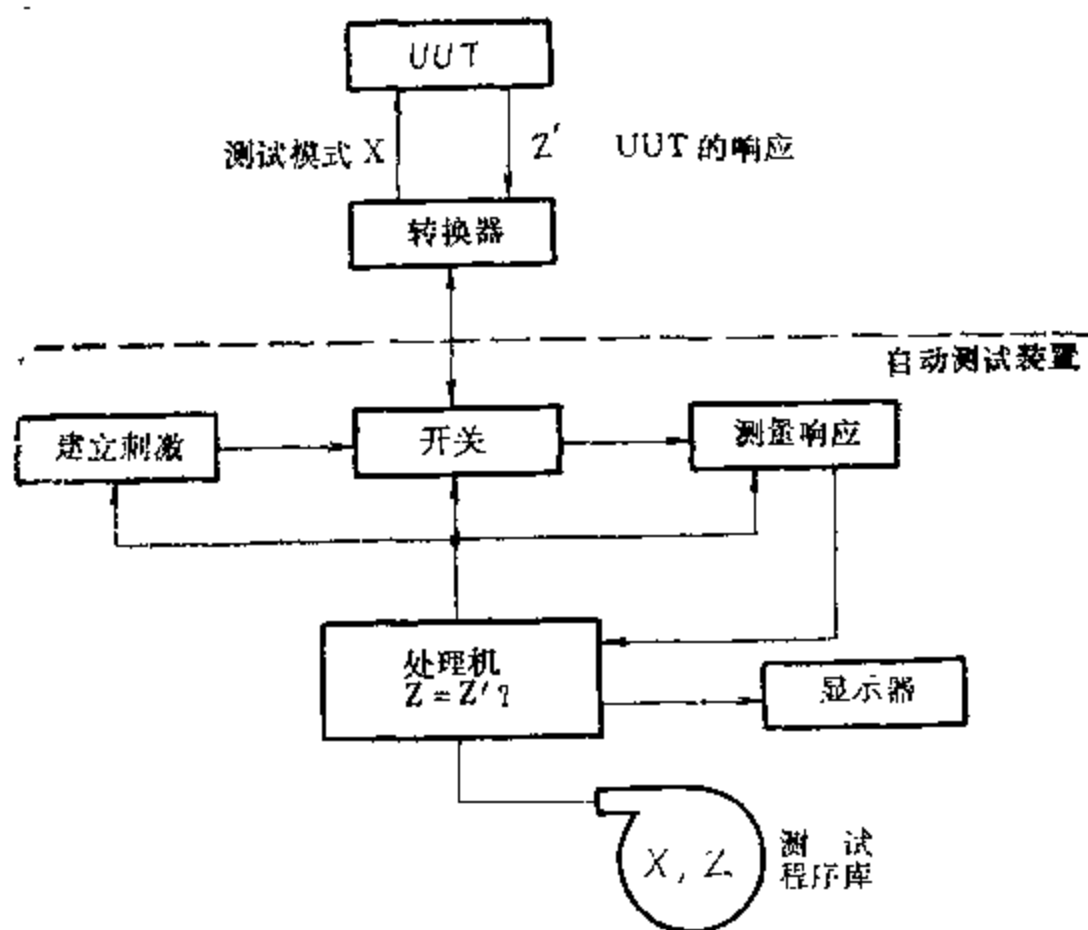


图 6.6 自动测试装置进行诊断

Z' 被送回到处理机与正常情况下的响应 Z 相比较。根据这些信息，自动测试装置能确定被测单元是否有故障。如果有故障，自动测试装置还能确定故障发生在何处。

那么这些测试模式及其响应是怎样得出来的呢？对复杂的计算机系统，用人工产生这些测试模式，其工作量是不可想象的。这些测试模式通常是用测试模式自动生成系统自动产生的。图 6.7 是一个典型的测试模式自动生成系统框图。对系统的输入包括要生成测试模式的电路的描述、要被测试的故障和初始信息。图中“模式生成器”的任务是根据电路的描述，对给定的故障产生它的测试模式及其响应。“故障模拟器”的作用是反过来对已算出的测试

模式确定它还可以测试哪些故障。当复盖率达到要求后，系统就建立故障检测和定位字典。最后根据这些信息生成测试程序，存贮在磁带中。

在这里，如何根据线路的描述，对给定的故障计算出它的测试模式及对应的响应，是一个中心问题。目前主要有两种算法：D 算法和布尔差分法。D 算法是启发式的方法，布尔差分法是解析的方法。从国内外实践来看，D 算法采用得比较多。

2.1-1 D 算法

在前面提到的例子中，我们是通过敏化一条单通路来求得测试模式的。这方法称为一维通路敏化法。但有的电路，用一维通路敏化法是求不出测试模式的。必须同时在几条通路上传播出错信号，才能求出测试模式。D 算法就是能同时敏化多条通路的方法，因此有时也称为二维通路敏化法。

为了便于用计算机来自动生成测试模式，通路敏化过程一般是用立方代数来形式化的。在具体介绍 D 算法之前，我们先来熟悉以下几个基本术语。

(1) 原立方 (Primary Cube)

设有一个组合线路单元 E 实现逻辑函数 f ， f 的原立方就是 f 和 \bar{f} 的原本蕴涵式的表格表示。图 6.8(a)表示电路元 E，图 6.8(b)为其实现的逻辑函数 f 。f 的原本蕴涵式为 $f = \bar{x}_1 x_3 + x_2$ ， \bar{f} 的原本蕴涵式为 $\bar{f} = \bar{x}_2 x_3 + x_1 \bar{x}_2$ 。E 的原立方就如图 6.8(c)所示。线 1，线 2，线 3 分别代表输入端 x_1 ， x_2 ， x_3 ，线 4 代表输出。表中每一行都称为一个原立方。 β_1 包括二个原立方，分别对应于 $f = \bar{x}_1 x_3 + x_2$ 中的二个原本蕴涵项。 β_1 的第一个原立方对应于第一项

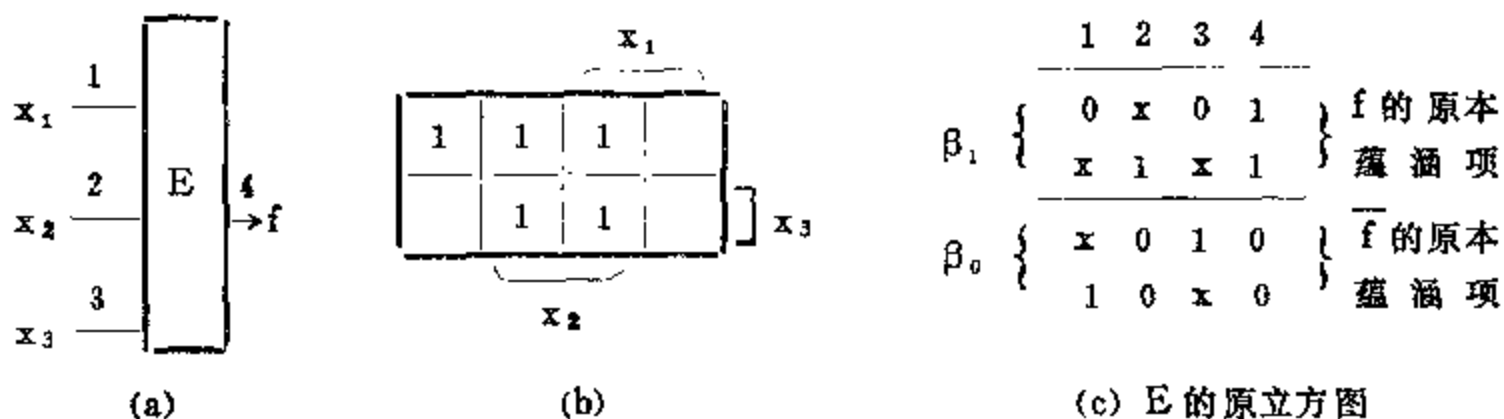


图 6.8 电路 E 及其原立方

$\bar{x}_1 x_3$ 。因为在这一项中 x_2 不出现，原立方在 2 下面是 x，x 表示不定值，可以是“0”，也可

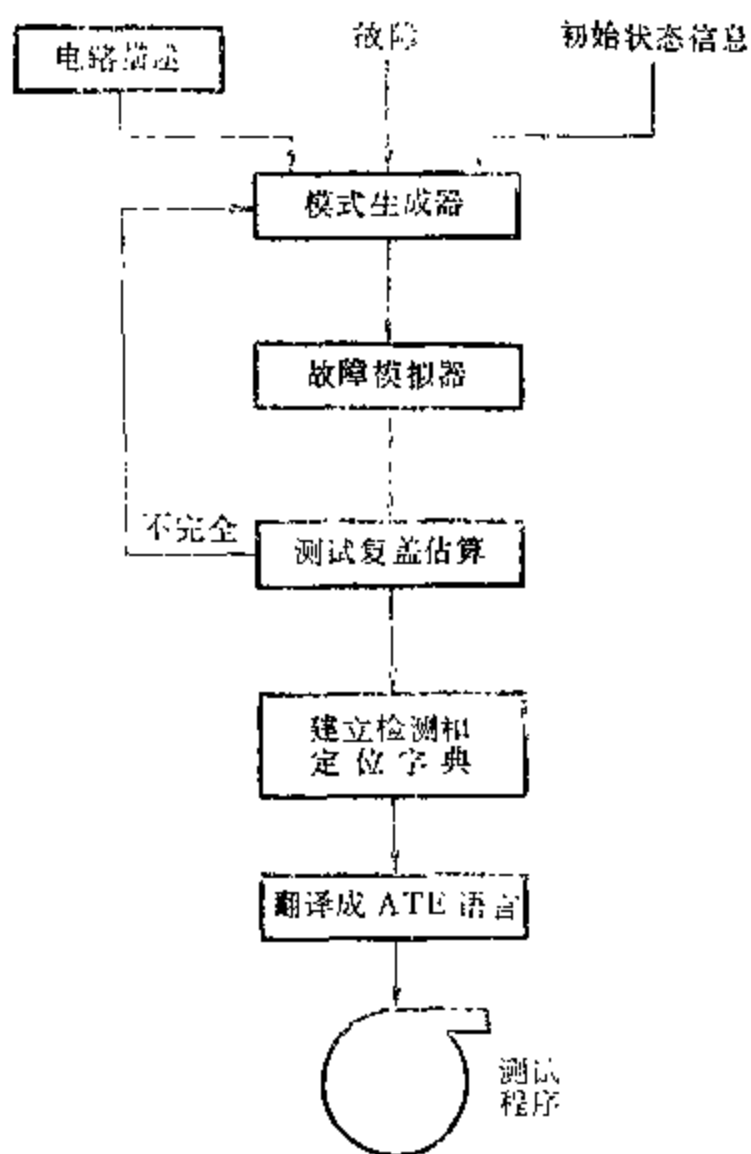


图 6.7 测试模式自动生成系统
(注) ATE 为自动测试装置

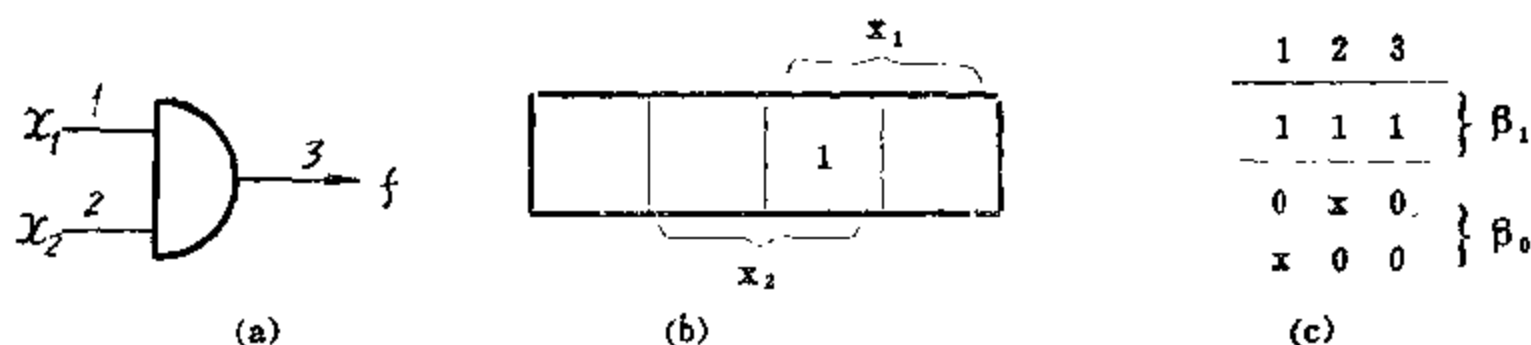


图 6.9 与门及其原立方

以是“1”。显然，原立方表实际上就是真值表的压缩形式，它用最少的输入信号简洁地表征了E的逻辑行为。图6.9(c)，图6.10(c)，图6.11(c)分别给出了与门、或门和与非门的原立方。

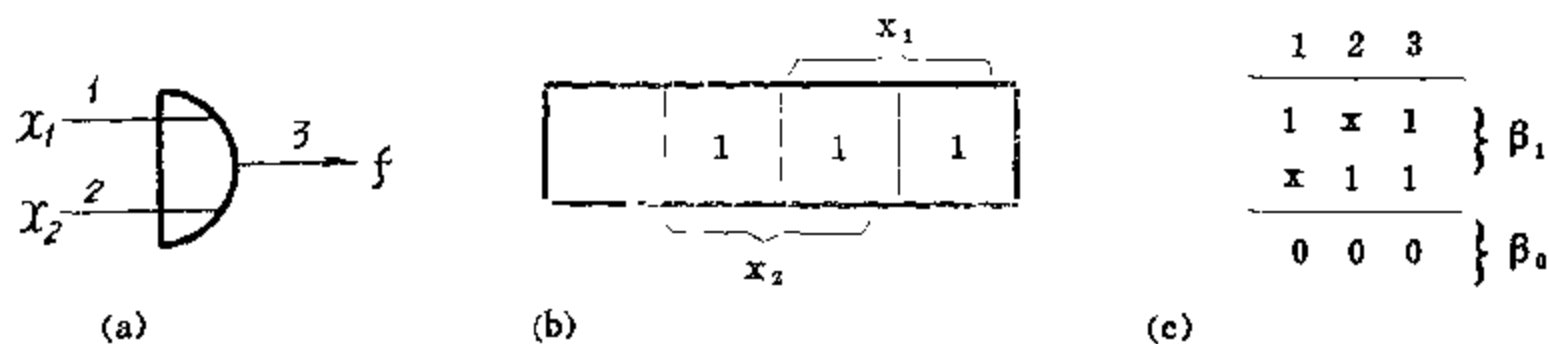


图 6.10 或门及其原立方

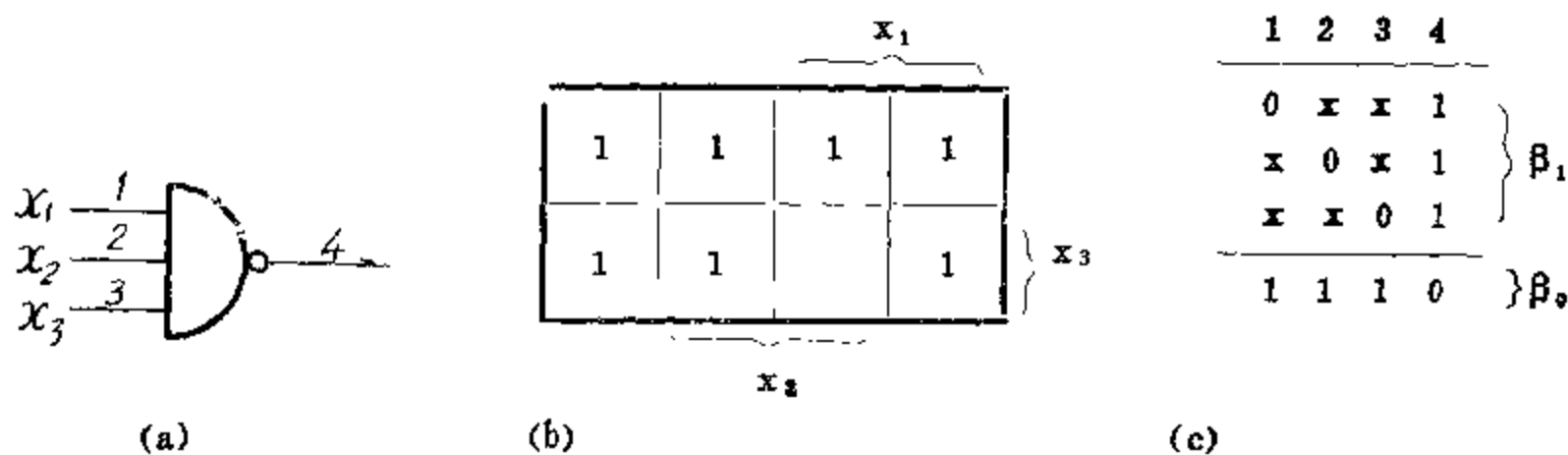


图 6.11 与非门及其原立方

(2) 故障原D立方

设线路单元E中有一故障 α ， α 的故障原D立方就是使E的输出端产生出错信号D或 \bar{D} 的最少输入条件。

在介绍故障原D立方的构成方法之前，先定义两个立方的交。一个立方相当于电路中各线的一种赋值，它在形式上是一个向量。立方 δ 和立方 ϵ 的交 $v = \delta \cap \epsilon$ 是这样：如果 δ 和 ϵ 的同一分量的值相等，则 v 的对应分量就取此值；如果 δ 和 ϵ 的同一分量的值中有一个为x，则 v 的对应分量就取另一个值作为自己的值，只要 δ 和 ϵ 有一对分量的值相反，则 v 就无意义，即 $\delta \cap \epsilon = \phi$ 。这时称 δ 和 ϵ 是不一致的。实际上 $\delta \cap \epsilon$ 表示的是同时满足 δ 和 ϵ 的电路各点的赋值， $\delta \cap \epsilon = \phi$ 表示 δ 和 ϵ 不可能同时满足。例如，如 $\delta = 0x10$ ， $\epsilon = x110$ ， $\mu = 1x1x$ ，则 $\delta \cap \epsilon = 0110$ ， $\epsilon \cap \mu = 1110$ ， $\delta \cap \mu = \phi$ 。

设E的原立方是 β_1 和 β_0 。E出故障 α 后，功能发生变化，实现的函数不再是 f ，而是 f_α 了。设函数 f_α 的原立方为 α_1 和 α_0 。（注意，这里 β_1 ， β_0 ， α_1 ， α_0 都是原立方的集

合) 把 β_1 中和 α_0 中的立方的输入部分相交 (如果相交结果不为 ϕ 的话), 就得到输出 D 的故障原 D 立方; 把 β_0 中和 α_1 中的立方的输入部分相交 (如果相交的结果不为 ϕ 的话), 就得到输出 \bar{D} 的故障原 D 立方。设图 6.8 的线路单元 E 出故障 α 后实现的 f_α 如图 6.12(a) 所示。 f_α 的原立方在图 6.12(b)。根据图 6.8(c) 和图 6.12(b), 可得出如图 6.12(c) 所示的故障原 D 立方。在这里, 第一个故障原 D 立方是由 β_1 的第一个原立方和 α_0 的第一个原立方相交而来的。第二个故障原 D 立方是由 β_1 的第二个原立方和 α_0 的第一个原立方相交而来的。第三个故障原 D 立方是由 β_0 的第二个原立方和 α_1 的第一个原立方相交而来的。

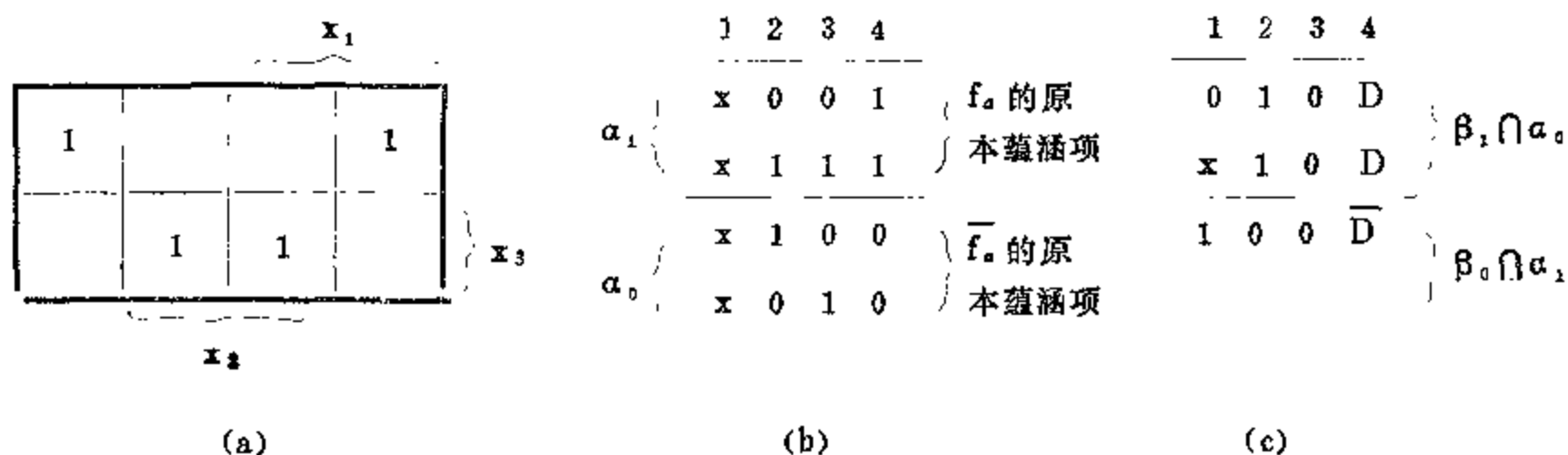


图 6.12 图 6.8 例的故障原 D 立方

同理, 对一个与门 (图 6.13(a)) 的输出端 “固定 0” 故障, 其故障函数 $f_\alpha = 0$, f_α 的原立方如图 6.13(b) 所示。参照图 6.9(c) 的 f 的原立方, 可得出图 6.13(c) 所示的故障原 D 立方。

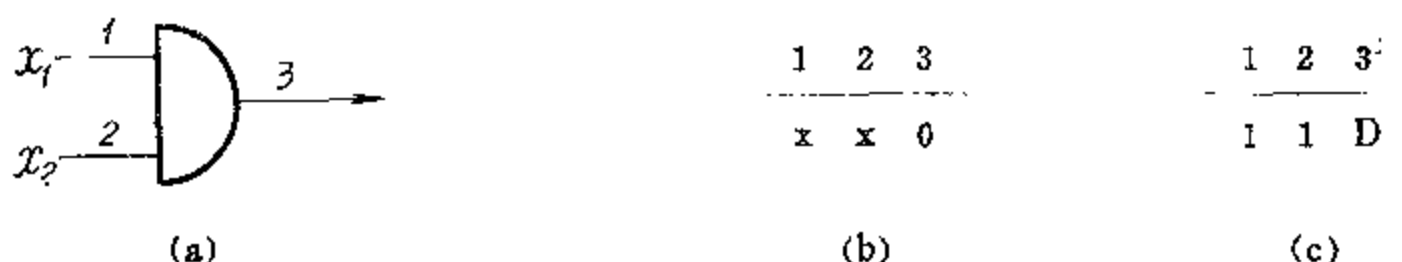


图 6.13 “与门” 输出 “固定 0” 故障原 D 立方

(3) 传播 D 立方

传播 D 立方 表示的是把 E 的一个或几个输入端上的出错信号传播到 E 的输出端的最小输入条件。

设线路单元 E 的原立方分别为 β_1 (输出为 “1”) 和 β_0 (输出为 “0”)。 E 的传播 D 立方可以这样得到: 或者把 β_1 中第 r 个输入端为 “0” 的输入立方 (即立方的输入部分) 和 β_0 中第 r 个输入端为 “1” 的输入立方相交 (第 r 个输入端不参加相交, 如果可以相交的话), 得出第 r 个输入端为 \bar{D} 、输出端为 D 的传播 D 立方; 或者把 β_1 中第 r 个输入端为 “1” 的原立方和 β_0 中第 r 个输入端为 “0” 的原立方相交 (第 r 个输入端不参加相交, 如果可以相交的话), 得出第 r 个输入端为 D 、输出端为 D 的传播 D 立方。

这里应注意以下两点。第一, 传播 D 立方总是成对出现的。它们的区别仅在于有出错信号的分量相反。就拿图 6.9 的与门来讲, β_1 的原立方和 β_0 的第一个原立方相交, 就能得出传播 D 立方, 如图 6.14(a) 的第一行所示。如 β_0 的第一个原立方和 β_1 的原立方相交, 就

得出如图6.14(a)的第二行所示的传播D立方。同理，考虑到 β_1 的原立方和 β_0 的第二个原立方之间的关系，就会得出图6.14(b)所示的传播D立方。第二，因为出错信号可能在多条通路上同时传播，所以应把输入端出现多个出错信号的情况考虑进去。在图6.9(c)中， β_0 的第一个原立方的线2处是x，如果把x看作“0”，就可以得出图6.14(c)的传播D立方（如果把 β_0 中的第二个原立方中的x看作“0”，也可以得出同样结果）。最后考虑到对偶性，与门的所有传播D立方可以归结为图6.14(d)。

1 2 3	1 2 3	1 2 3	1 2 3
D 1 D	1 D D	D D D	D 1 D
\overline{D} 1 \overline{D}	1 \overline{D} \overline{D}	\overline{D} \overline{D} \overline{D}	1 D D
			D D D
(a)	(b)	(c)	(d)

图 6.14 与门的传播D立方

D算法的基本思路如下：先在有故障的线路单元E的对应故障原D立方中任选一个，这样E的输出为D或 \overline{D} ，其输入端是一些“0”、“1”或x。这时，凡是以E的输出作为输入的单元组成了D—前沿。在任何时候，电路中输入端有出错信号D或 \overline{D} 而输出端为x的单元的集合称为D—前沿。任选D—前沿中的一个单元，利用它的传播D立方可以把D或 \overline{D} 再向前驱赶一次，传播到它的输出端。这一过程称为D—驱赶。这样，又形成了新的D—前沿。如此，一步一步地把D—前沿向前推进，直至把D或 \overline{D} 驱赶到电路的输出为止。每当进行一次D—驱赶后，电路中某些原来未定的值x变成了确定的值“1”或“0”。凡是与这新确定的值相关联的单元，如果根据当前电路状况能唯一确定出其它一些输入或输出的值，都应根据它的原立方把这些输入或输出端的值确定下来，这一步称为蕴涵。如果有一根线上原来值已为“1”（或“0”），而蕴涵时又要将它的值定为相反的值“0”（或“1”），这就发生了不一致。不一致说明了最近的一次D—驱赶是不对的，从而宣布无效，控制又返回到最近的一次D—驱赶之前的状态，更换一种传播D立方试试，这也称为回头。如果该单元所有的传播D立方都试过，都出现不一致，则控制就再返回到前一个选择点：在D—前沿中再取下一个单元来进行D—驱赶。D算法运行过程就是这样一个不断地搜索、不断地回头的启发式过程。因为D算法实际上可以对D—前沿中的单元进行并行D—驱赶，所以它可以敏化多通路。这是一个功能很强的算法。

2.1-2 布尔差分法

设F是n个变量 x_1, x_2, \dots, x_n 的一个布尔函数，它对某一变量 x_i 的布尔差分 dF/dx_i 定义为：

$$\frac{dF(x_1, \dots, x_i, \dots, x_n)}{dx_i} = F(x_1, \dots, x_i, \dots, x_n) \oplus F(x_1, \dots, \overline{x_i}, \dots, x_n)$$

其中， \oplus 表示异或运算。根据此定义，可以知道：若 $dF/dx_i = 1$ ，则 x_i 取相反值时，F也取相反值；反之亦然。如果某一电路有n个输入 x_1, x_2, \dots, x_n ，它实现函数 $F(x_1, \dots, x_n)$ ，若 $dF/dx_i = 1$ ，则在 x_i 到F输出之间必存在敏化通路，将 x_i 处的出错信号传到电路的输出

端。

$$\begin{aligned}\frac{dF}{dx_i} &= F(x_1, \dots, x_i, \dots, x_n) \oplus F(x_1, \dots, \overline{x_i}, \dots, x_n) \\ &= F(x_1, \dots, 1, \dots, x_n) \oplus F(x_1, \dots, 0, \dots, x_n)\end{aligned}$$

这就是说, dF/dx_i 是 $n-1$ 个变量 $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ 的布尔函数, 它与 x_i 无关。

在图 6.15 中, Y 表示电路内部的某一根线。如要测试 Y 的“固定 0”故障, 应使 Y 的原来值为“1”。设 Y 对 x_1, x_2, \dots, x_n 的关系可以用布尔函数 $Y = Y(x_1, x_2, \dots, x_n)$ 来表示。 F 输出可以表示为 Y, x_1, x_2, \dots, x_n 的布尔函数 $F = F(Y, x_1, \dots, x_n)$ 。那么, 满足方程



图 6.15

$$Y(x_1, \dots, x_n) \cdot \frac{dF(Y, x_1, \dots, x_n)}{dY} = 1$$

的输入向量 (x_1, x_2, \dots, x_n) 必是此故障的测试模式。这里的“ \cdot ”是“与”运算。因为此输入向量将使 Y 取“1”值, 同时又能敏化从 Y 到 F 的通路。同样, 若求 Y 的“固定 1”故障的测试模式, 只要解方程

$$\overline{Y}(x_1, \dots, x_n) \cdot \frac{dF(Y, x_1, \dots, x_n)}{dY} = 1$$

就行了。

布尔差分法已在其它课, 如“数字逻辑”课中讲过, 这里不再详细叙述。总之, 布尔差分法是数学上严谨、概念上直观简单的解析方法。采用布尔差分法首先是对故障端建立测试方程, 这个方程就是描述被测网络的布尔函数对故障端变量的布尔差分。解这个方程求得完全测试集。解方程, 实际上是对两个布尔函数作一次异或运算, 其中一个表示正常电路的函数, 另一个是表示电路有故障的函数。最后乘上故障端的条件就得到了该故障的所有可能的测试模式。

布尔差分法虽然在理论上比较完善, 但是, 当网络较大时, 其测试方程的解析式将变得十分冗长, 求解运算也相当繁琐, 因而, 使得它需要的存储空间很大, 计算时间长, 所以目前的测试模式自动生成系统一般多采用 D 算法。

从 § 2.1-1 和 § 2.1-2 可以看出, 这二种方法是以电路中只有单故障为前提的。因为出单故障的可能性比出多故障的可能性大得多, 所以在实际应用时, 假定电路中仅发生单故障还是可行的。

2.2 微诊断法

故障定位测试法能把故障定位到一个门, 一根线。但随着电路规模的增大, 诊断过程的复杂性也迅速增加。即使是一台小型计算机, 想用故障定位测试法一下子把故障定位到门一级, 就是能够的话, 其计算的复杂性也是无法对付的。因此需要有一种适应不同分辨率要求的诊断方法。我们可以先把故障定位到子系统或模块一级, 然后对出故障的子系统或模块进行测试。把故障定位到某一子系统, 这种诊断方法一般是把被诊断的对象看作为一个“黑箱”, 它只涉及到此对象的输入和输出之间的关系, 而无需知道它内部结构的细节。这就是

所谓的功能测试法。另外，随着微电子学的发展，计算机的元器件的集成度不断提高，功能测试法就更加显得重要。

用以机器语言写的检查诊断程序来进行诊断的方法，就是一种功能测试法。它是利用机器指令的功能来对系统的某些部件进行检测的。但由于一条机器指令的正确实现，往往要涉及到许多部件。因此用机器指令写的诊断程序的分辨率较低，而且运行这种检查诊断程序的时间也较长。另外，在检查诊断程序运行时，要求系统中能正常工作的部件的数量较大，否则检查诊断程序就不能正确地执行。也就是说，它要求系统的硬核比较大。

近年来，国内外发展了一种在微程序设计的计算机中采用的诊断方法——微诊断法。微诊断法是用微指令的功能来对系统进行诊断的。所以也叫微指令功能测试法。微诊断程序类似于机器语言组成的检查诊断程序。所不同的是它是在微程序设计的机器中用更低一级的语言，即微指令组成诊断微程序，进行微命令的功能检查，间接找出故障所在。一般微命令所定义的操作只涉及较少的部件，所以，它可以把故障分辨得很细。它的运行时间显然要短得多。

微诊断法还有一个优点。一般的机器语言组成的诊断程序往往是停机之后由操作员启动的。而微诊断程序是由微指令组成，一般存放在控制存储器中。用一条特定的指令，就可以调用微诊断程序。因此它可以穿插在用户程序之中，在用户程序完成了某一任务后，自动启动微诊断程序。这就是动态微诊断。这样就带来了如下二个好处。一是把微诊断技术、指令复执、微指令复执和程序卷回技术结合起来，就能有效地度过一些暂时性故障。二是对于要求有自修复能力的容错计算机来讲，它提供了动态诊断和在故障定位后实现动态切换的可能性。

在检查某一功能单元 F_i 时，通过执行一系列微操作，先在与 F_i 对应的寄存器里形成测试数据，然后使 F_i 单元执行一定的微操作，最后把所得到的结果与正常情况下应得到的结果相比。在系统诊断时，往往采用引导的策略。引导的方法，有时也称为“滚雪球”的方法。

引导过程是这样的。开始时，用硬核对系统的某一小部分进行测试。如果没有发现故障，就以这部分电路和硬核为基础来对系统中没有测试过的另一部分进行测试。这样一步一步地扩展被测试的区域。如果从开始直到第 i 层都没有发现故障，而在测试第 $i+1$ 层 A_{i+1} 时发现故障，则可以断定故障发生在 A_{i+1} 区域中，见图6.16所示。

有的微程序设计的机器，它有二段微诊断程序：动态微诊断程序和静态微诊断程序。其中，静态微诊断程序用来诊断运控部分的固定性故障。它作为一种维护手段，往往是在用户程序终止

后，由操作员启动的。因此与插入在用户程序中的动态微诊断程序相比，在时间上的要求没有那么严格。所以在由硬核一步一步向外扩展被测试区域时，每一层所涉及到的电路可以比

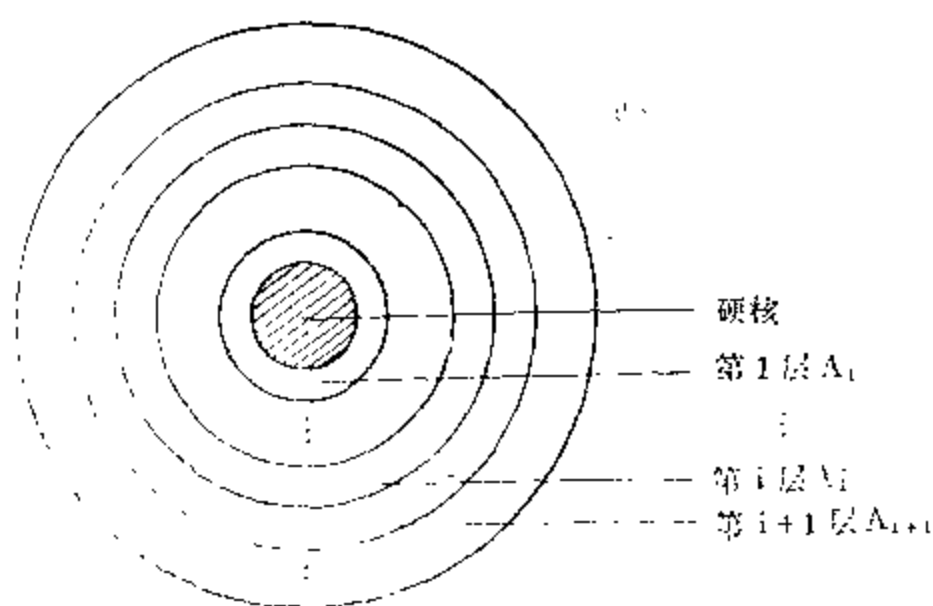


图 6.16 由硬核逐层扩展测试

较少，从而可以达到比较高的分辨率。故障复盖率也可以做得比较高。动态微诊断程序因为是插在用户程序中运行的，对它的要求和静态微诊断的不同。第一，要求它运行的时间要尽可能短，否则就可能丢失某些实时信息。所以，它的故障复盖率就要受到限制，往往只能对某些重要的或容易出故障的组件进行诊断。由于时间要求尽可能短，一般就要求诊断时，让一组测试数据尽可能通过较多的部件，即数据通路要尽可能长。第二，动态微诊断应能保护机器的现场。第三，它应能区别出暂时性故障和固定性故障。因为对这两种故障，机器应作的处理是不同的。如果是暂时性故障，机器应从最近的一个卷回点开始进行程序卷回。如果是固定性故障，机器或者停机，或者自动切换掉故障部件，再进行程序卷回或指令复执，实现容错。

还应指出的是，微程序设计的计算机，利用微诊断还可以通过前述指令复执来度过暂时性故障。因为执行一条机器指令相当于调用一段微程序，那么在微程序中插入某些用于检测的微命令，就可以在指令执行完毕前检测到故障。一旦发现故障，就重取这条指令，重复执行若干次。如果不再出错，就认为复执已成功，机器也就度过了一次暂时性故障，原先的程序可以继续运行。

这种复执的思想还可以深入到微指令一级，这就是微指令复执。

以上介绍了二种故障诊断的方法。总的说来，故障定位测试法分辨率高，但计算复杂。微诊断法的分辨率虽然比故障定位测试法的低，但它作为一种功能测试法，用于系统诊断比较有效。另外，它可以用于多故障情况，要求的硬核比较小。因此，这二种方法不能互相代替，只能相辅相成。

一般来说，对于静态冗余系统，不宜在正常运行过程中进行诊断。这是因为诊断意味着要查出故障的有无和所在的范围，但静态冗余却起了“掩蔽”故障的作用。两者是矛盾的。

§ 3 误差校正码

已被广泛采用的奇偶校验码可以用来检测信息传送中的一位错。如果添加更多的奇偶校验位，可以检测同时出现的多位错，或对出错进行精确的定位，从而提供了自动纠错的条件。这就是误差校正码的作用。误差校正码即为前面提过的纠错码。

尽管误差校正码原先是为通信系统研制的，但是近年来在计算机系统的设计中已有重要应用。主要是用于提高内存和外设的工作可靠性，其次在运算部件中也有应用。采用误差校正码可以检测和校正暂时性的故障，也可以检测和校正固定性的故障。但误差校正码作为一种信息冗余手段，事实上起到“掩蔽”故障的作用。因此采用误差校正码为排除暂时性故障能起积极作用，但对检测固定性故障却起消极的作用。通常，计算机系统中发生暂时性故障的概率要比发生固定性故障的概率大得多，因此，一般说来，采用误差校正码的有效性还是无可非议的。

计算机系统中，就硬件来说，一般是内存不及中央处理部件可靠，而外设又不及内存可靠。因此在外设和内存方面，近年来在广泛采用奇偶检测的基础上已采用误差校正码来“掩蔽”故障，实现容错。

存贮器在读出和写入时，可能出错，其中以出现单错的概率最大。如果是只读存贮器，

读出时可以用奇偶位来校验，一旦发现读错，通过多次重读是可能纠错的。但磁心存贮器是按破坏性读出方式工作的，读出后原信息已被破坏，如读出时出错，重写时就会把错误的信息写到磁心存贮器中。所以用重读的方法是不能进行纠错的。近年来国内外已广泛采用能自动纠正单错、发现双错的推广海明纠错码来提高磁心存贮器的可靠性。它是线性码的一种。

在外设方面，就以磁带机为例，其优点是存贮量大，其缺点是可靠性差。考虑到磁带镀层的缺陷和磁头或磁带上可能有的微小尘埃往往会引起某条道上出现多位并发性错误，如果用循环冗余码（CRC 码，Cyclic Redundancy Code）校验的方法，可以将一条道上的多位错依次全部纠正过来。CRC 码是循环码的一种。下面介绍线性码和循环冗余码的原理和应用。

3.1 线性分组码

3.1-1 线性分组码及其生成矩阵

前述纠错码的传送和纠错过程可用图 6.17 的模型来描述。信号源每次发出 k 位二进制信息 $m = (m_1, m_2, \dots, m_k)$ ，经编码器按一定的线性规则在它后面加上 r 位校验位（也叫

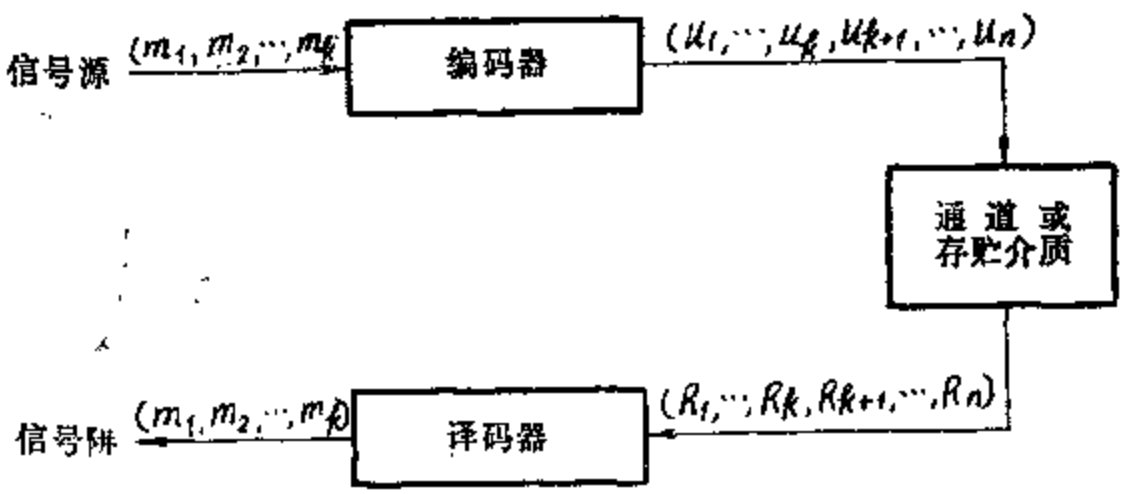


图 6.17 误差校正码的检测和纠错模型

冗余位）组成一个长为 $n = k + r$ 位的二进制序列 $u = (u_1, u_2, \dots, u_n)$ 。这种形式的编码我们称之为系统线性码。因为信息是分成一组一组地处理，所以又称它为线性分组码，也称线性 (n, k) 码。其结构如图 6.18 所示。可用 $R_c = k/n$ 表示编码中信息位所占的比例，称为编码效率，简称码率。按图 6.18 这种结构编出的长为 n 位的二进制序列称为码字， n 为码长，其中各位分量称为码元。 n 位长的二进制序列（也称 n 重）共可有 2^n 种不同的组合，

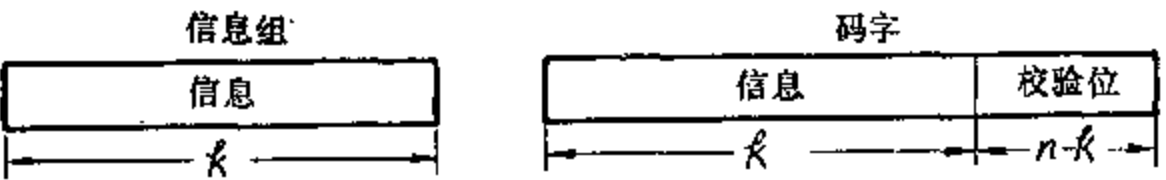


图 6.18 线性分组码结构

而 k 位信息总共只能有 2^k 种不同的组合，所以我们只须从 2^n 个 n 重中选 2^k 个作为许用码字，剩下的 $2^n - 2^k$ 个 n 重则为禁用码字。当然，我们一般所说的码字都是指许用码字。经编

码器输出的码字通过传输通道或存贮介质后可能由于各种原因产生错误,收到的编码变成 $R = (R_1, R_2, \dots, R_n)$ 。如果收到的编码 R 与发送的编码 u 不一致,即 $R \neq u$ 时,则表明信息在传送过程中出现了错误。在一定限度上,纠错码能发现有错,并能把出错的位纠正过来。译码器的任务就是把收到的 R 还原为 u ,再从 u 和 m 的对应关系得到原来发送的信息 m 。

由此可见,分组码的编码问题实际上就是如何按一定规则从 2^n 个 n 重中选取 2^k 个 n 重作为码字的问题。也就是建立起 k 位信息序列与 n 位编码序列之间的一一对应关系:

$$\begin{array}{ccc} \underbrace{(m_1, m_2, \dots, m_k)}_{k \text{ 位信息序列}} & \xrightarrow{\text{编码}} & \underbrace{u_1, u_2, \dots, u_k}_{k \text{ 个信息位}}, \underbrace{u_{k+1}, \dots, u_n}_{n-k \text{ 个校验位}} \end{array}$$

下面我们用数学语言来描述这种关系。

设 U 是 n 位二进制数码的集合,即

$$U = \{(u_1, u_2, \dots, u_n) | u_i \in \{0, 1\}, i = 1, 2, \dots, n\}$$

若 U 中某一元素 $u = (u_1, u_2, \dots, u_k, u_{k+1}, \dots, u_n)$ 是一个码字,而 $m = (m_1, m_2, \dots, m_k)$ 是一信息组,则应有

$$u = (m_1, m_2, \dots, m_k) \begin{bmatrix} 1 & 0 & \dots & 0 & p_{11} & p_{12} & \dots & p_{1(n-k)} \\ 0 & 1 & \dots & 0 & p_{21} & p_{22} & \dots & p_{2(n-k)} \\ \vdots & & & \vdots & \vdots & & & \vdots \\ 0 & \dots & 1 & 0 & & & & \\ 0 & \dots & 0 & 1 & p_{k1} & p_{k2} & \dots & p_{k(n-k)} \end{bmatrix} \quad (6.3-1)$$

作此矩阵乘法可得:

$$u_i = m_i \quad i = 1, 2, \dots, k$$

$$\text{而 } u_{k+j} = p_{1j}m_1 + p_{2j}m_2 + \dots + p_{kj}m_k \quad j = 1, 2, \dots, (n-k) \quad (6.3-2)$$

注意,在本节中遇到的‘+’都指模2加法运算。

由此可见,该码字前 k 位数字正好是信息的 k 个数字,后面的 $n-k$ 位是 k 个信息数字的线性函数。而后面这 $n-k$ 位正是校验位,由此可找到信息位和校验位的线性关系。(6.3-2)式也称为校验方程。根据编码规则,只要给定一个信息码组 m ,就可以得出对应的编码 u 。我们将简写成

$$u = m \cdot G \quad (6.3-3)$$

我们把矩阵 G 称为生成矩阵。生成矩阵 G 又可写成如下形式

$$G = [I_k P] \quad (6.3-4)$$

其中 I_k 是 $k \times k$ 阶单位方阵, P 为 $k \times (n-k)$ 阶矩阵, G 是 $k \times n$ 阶矩阵。这样,编码问题就归结为怎样选择满足一定要求的生成矩阵 G 的问题。而最重要的要求是在一定的编码效率下,纠错或检错能力应尽可能强,并且容易实现。要详细讨论纠错、检错问题,需要搞清海明距离和海明重量这两个重要的概念。

3.1-2 距离、重量和纠错能力

两个码字之间对应位上取不同值的个数称为海明距离,简称距离。

若给定两个码字 $u = (u_1, u_2, \dots, u_n)$

$$u' = (u'_1, u'_2, \dots, u'_n)$$

则 u 和 u' 之间的海明距离定义为

$$d(u, u') = \sum_{i=1}^n (u_i + u'_i) \quad (6.3-5)$$

在码字集合中, 任意两个码字之间距离的最小值称为最小距离, 记为 d_{\min} 。最小距离是衡量一个编码的抗干扰能力大小的标准。码的最小距离越大, 表明码字集合中任意两个码字之间的最小差别越大, 因此, 抗干扰能力就越强。

一个码字的海明重量是指该码字中非零码元的个数, 简称重量, 记为 W 。

在二进制情况下, 码字的重量即是该码字中 1 的个数。若有一码字 $u = (u_1, u_2, \dots, u_n)$, 其重量为

$$w(u) = \sum_{i=1}^n u_i \quad (6.3-6)$$

显然, 任意两个码字之间的距离就等于它们之和的重量, 即

$$d(u, u') = w(u + u') \quad (6.3-7)$$

由于线性码的封闭性, 任意两个码字之和得到另一个新码字, 所以任意两个码字之间的距离就等于另一个码字的重量。从此可以得出一个重要的结论: 一个线性码的最小距离就等于所有非零码字重量的最小值。

对于前述的 u 和 R , 若用向量表示, 则 R 可看作是 u 和错误向量 e 之和, $R = u + e$, $e = (e_1, \dots, e_n)$ 也称为错误模式。根据异或运算的性质, $e = R + u$ 。所以当 $R_i \neq u_i$ 时, $e_i = 1$, 表示在 R 的第 i 个分量有错。因此 $W(e)$ 表示出错的个数, 也是 R 和 u 之间的距离。但是 R 分解成的码字和错误模式并不是唯一的, 也有可能 $R = u' + e'$, 这里 u' 是另一个码字, e' 是另一个错误模式。这时该把 u 和 u' 哪一个作为在实际传输过程中的码字呢? 一般而言, 在传输过程中出错位数较少的概率比出错位数较多的概率要大。因此, 在纠错时, 总是把与 R 的距离最小的码向量作为实际传输的码向量。称此为最似然译码原则。根据此原则就能得出下列重要结论:

对一个线性 (n, k) 码, 其极小距离为 d_{\min} ,

(1) 若 $d_{\min} \geq e + 1$ 成立, 它可以检测 e 个错;

(2) 若 $d_{\min} \geq 2t + 1$ 成立, 它可以纠正 t 个错;

(3) 若 $d_{\min} \geq e + t + 1 (e > t)$, 它可以同时检测 e 个错和纠正 t 个错。

因为任何二个码向量之间的距离大于等于 d_{\min} , 如发生 e 位错, $d(R, u) = w(E) = e$, 而 $e < d_{\min}$, 所以 R 必定不是码字。在检测时, 判定一下收到的向量 R 是不是码字, 如果 R 不是码字, 就说明已出错, 如果 R 是码字, 也可以肯定没有出错。第一个结论正确。

设 u 是被传输的码向量, V 是另外一个码向量, R 为接收到的向量。 u 、 V 、 R 之间满足下列不等式

$$d(V, R) + d(R, u) \geq d(u, V)$$

根据 (2) 给定的条件得 $d(u, V) \geq d_{\min} \geq 2t + 1$ 。因发生了 t 位错, 所以 $d(R, u) = t$,

于是 $d(V, R) + t \geq 2t + 1$, 最后 $d(V, R) \geq t + 1$ 。这说明, 接收向量 R 与码向量 u 的距离小于 R 与任一其它码向量的距离。这样, 在纠错时就能从 R 正确地还原出 u , 达到纠错的目的。第二个结论也正确。

至于第三个结论是第一个和第二个结论的直接推论。因为 $e > t$, 从 $d_{\min} \geq e + t + 1$, 能推出 $d_{\min} \geq 2t + 1$; 又 $t > 0$, 从 $d_{\min} \geq e + t + 1$, 同时也能推出 $d_{\min} \geq e + 1$ 。有了前二个结论, 第三个结论也是正确的。

由此可见, 一个线性码的极小距离 d_{\min} 与纠错能力直接有关, d_{\min} 越大, 纠错能力越强。

前面已讲到, 要检测错误, 就得判定 R 是否是一个码向量, 要纠正错误就得知道重量最小的错误模式 E 。本小节解决第一个问题, 下一小节解决第二个问题。

3.1-3 一致校验矩阵与伴随式

现在我们以 $(7, 4)$ 分组码为例, 从另外一个角度建立信息位与校验位之间的关系。

假设码长 $n = 7$; 信息位 $k = 4$, 为 $u_7 u_6 u_5 u_4$; 校验位 $r = 3$, 为 $u_3 u_2 u_1$ 。假若我们已求得校验位与信息位的线性关系如下 (同上一节, 这里 “+” 为 “ \oplus ”):

$$\begin{cases} u_3 = u_7 + u_6 + u_5 \\ u_2 = u_6 + u_5 + u_4 \\ u_1 = u_7 + u_6 + u_4 \end{cases}$$

即 $u_7 u_6 u_5 u_4 u_3 u_2 u_1$ 若是一个码字必须满足以下校验方程:

$$\begin{cases} u_7 + u_6 + u_5 + u_3 = 0 \\ u_6 + u_5 + u_4 + u_2 = 0 \\ u_7 + u_6 + u_4 + u_1 = 0 \end{cases}$$

写成矩阵形式为:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_7 \\ u_6 \\ u_5 \\ u_4 \\ u_3 \\ u_2 \\ u_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (6.3-7)$$

或简记为:

$$H \cdot u^T = 0^T \text{ 或 } u \cdot H^T = 0 \quad (6.3-8)$$

其中 $u = [u_7 u_6 u_5 u_4 u_3 u_2 u_1]$, u^T 是 u 的转置矩阵,

$0 = [0 \ 0 \ 0]$, 0^T 是 0 的转置矩阵。

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} = [P \cdot I_r] \quad (6.3-9)$$

H 被称为一致校验矩阵, 这里的 P 为 $r \times k$ 阶矩阵, I_r 为 $r \times r$ 阶单位方阵。 H 矩阵有 r 行, 可以建立 r 个线性方程式, 求得 r 个校验位, 当然这 r 个线性方程应该是独立的, 彼此线性

无关的。也就是说，H 矩阵中每行之间应彼此独立，线性无关。把 H 矩阵写成 (6.3-9) 式的 $[P \cdot I_r]$ 形式，就是为便于查看其行与行之间是否线性无关。写成式 (6.3-9) $[P \cdot I_r]$ 形式的 H 矩阵称为 H 的典型阵。校验位不放在信息位后面，可在任意码元位置，为了与典型阵形式的 H 校验矩阵构成的编码相区别，称之为非系统码。由式 (6.3-8) 可知 u 中各元素是满足由 H 矩阵所确定的 r 个线性方程式的解。反之，若 u 中各元素是一个码字的话，它必定满足 H 矩阵所给定的 r 个线性方程式，即满足 r 个信息位与校验位的关系式。由此可知，只要给出一致校验矩阵 H，信息位和校验位的关系就完全确定，编码问题也就解决了。H 矩阵不但解决了编码问题，同时也可解决译码问题。在接收端收到的信息 R 是不是一个码字，也可以用 H 校验矩阵验一验是否满足 $R \cdot H^T = 0$ ，若满足此条件， $R = u$ 是一个码字，否则 R 是一个禁用码字。也就实现了检错。

前面我们讲过给定一个信息码组 m ，由生成矩阵 G 可以给出编码 u ，并且也能给出信息位和校验位的关系。这里的一致校验矩阵 H，也能给出校验位和信息位的关系。那么，生成矩阵和 H 矩阵之间有什么关系呢？我们再来看由生成矩阵 G 生成码字的公式 $u = m \cdot G$ 为了叙述的方便，我们令

$$G = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix} \quad g_i \text{ 为行向量}, \quad i = 1, 2, \dots, k$$

则

$$u = (m_1, m_2, \dots, m_k) \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix}$$

若给定的信息 m ，其中只有某一位 $m_i = 1$ ，其余各位全为“0”，则

$$u = g_i, \quad i = 1, 2, \dots, k$$

即说明生成矩阵 G 中每一行都是一个码字。所以它应该满足

$$H \cdot G^T = 0^T \text{ 或 } G \cdot H^T = 0 \quad (6.3-10)$$

因此，只要得到了 H 矩阵，G 矩阵也就能求出，反之亦然。从 (6.3-10) 式可看出生成矩阵 G 是满足一致校验矩阵中 $r = n - k$ 个线性方程式的解；反过来，若把 G 看成一个校验矩阵，H 就是另外一种编码的生成矩阵。所以，如果用 G 生成矩阵得到是 (n, k) 码的话，那么由 H 作生成矩阵得到的就是 $(n, n - k)$ 码。所以 (n, k) 码总是对偶出现的。

一致校验矩阵 H，不仅能用于编码检错，同时也能由它找到纠错的办法。

在接收端收到的信息为 R，先检查一下它是否满足 $H \cdot R^T = 0^T$ 的条件。若满足，可以确信收到的 R 是一个码字；若不满足，则表明传输过程中出现了差错。出现了差错的 R 是如何得来的？如果我们假设错误信号是 $e = e_1 e_2 \dots e_n$ 发送来的正确编码是 $u = u_1 u_2 \dots u_n$ ，那么，我们收到的 $R = u + e = R_1 R_2 \dots R_n$ ，其中 $R_i = u_i + e_i$ ，现将 R 代入 $H \cdot R^T$

$$H \cdot R^T = H \cdot (u + e)^T = H \cdot u^T + H \cdot e^T$$

因为 u 是正确的码字， $H \cdot u^T = 0^T$ ，所以

$$H \cdot R^T = H \cdot e^T$$

$$\text{我们令} \quad H \cdot e^T = S^T \quad \text{或} \quad S = e \cdot H^T \quad (6.3-11)$$

我们称 S 为伴随式。伴随式只与错误模式 e 有关，而与码字本身无关。伴随式是从接收到的有差错信息得到的，是已知的，若能从伴随式中找出错误模式 e ，就能得到正确的码字 $u = R - e$ ，纠错的任务就完成了。下面分析如何从伴随式 S 得到错误模式 e 。

还以 $(7, 4)$ 码作为例子。若收到的 $R = 1110011$ ，出现的错误模式 $e = 0100000$ ，其一致校验矩阵为

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

计算伴随式如下：

$$S^T = H \cdot R^T = H \cdot e^T$$

$$= \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\therefore \quad S = [1 \ 1 \ 1]$$

由此可见，当错误模式中第几位有错误时，其伴随式 S 就正好是 H 矩阵中的第几列。反过来，当我们从收到的信息 R 计算出伴随式 S ，看它与 H 矩阵哪一列相同，就可以断定错误发生在哪一个对应位上。从而能得到错误模式。

综上所述，纠错码的编码传送、译码纠错整个过程是：对给定的信息码 m ，加上由生成矩阵或一致校验矩阵 H 确定的校验码，得到发送的编码 u ，在接收端收到编码 R ，计算伴随式 S ，看伴随式 $S = 0$ ？若 $S = 0$ ，则收到的 R 就是一个码字；若 $S \neq 0$ ，则 R 不是一个码字，查看 S 与 H 中哪一列相同，从中找出错误模式 e ，恢复原来的编码 $u = R - e$ ，在二进制中也可作 $u = R + e$ ，即实现了纠错。

3.1-4 海明码与推广海明码

由上一小节的分析可知，要能纠正一个码字中所有单个错误，则对应单个错误模式的每一个伴随式都不应相同。在长度为 n 的码字中，发生一个错误的模式有 n 种，伴随式也得有 n 种不同的组合（其中不包括对应正确编码伴随式为全“0”的一种）。伴随式即为 H 矩阵中的列向量， H 中共有 $r = n - k$ 行对应 r 个校验位方程，所以伴随式是一个 r 位的二进制数序列，它最多有 2^r 种不同的组合。所以，要求能纠 n 位长码字中所有单个错误必须满足：

$$2^r \geq n + 1 \quad \text{其中 } r = n - k \quad (6.3-12)$$

同理，要能纠正一个码字中所有两位以内的错误，则必须满足：

$$2^r \geq C_n^0 + C_n^1 + C_n^2 = 1 + n + \frac{n(n-1)}{2} \quad (6.3-13)$$

依此类推，若要求能纠正所有小于和等于 t 位错误必须满足：

$$2^r \geq C_n^0 + C_n^1 + C_n^2 + \dots + C_n^n = \sum_{i=0}^n C_n^i \quad (6.3-14)$$

海明码就是能纠正所有单个错误的线性分组码。(7, 4) 码就是一个海明码。它的一致校验矩阵有三行 7 列。三行对应确定三个校验位的方程, 7 列是对应 7 位码字上任一位出错误的伴随式。三重序列除了全 0 的三重外也只有这七种。虽然纠单个错误的海明码的 H 矩阵中列的次序是可以任意安排的, 但在实用中为了译码实现简单, 一般都采用典型阵型式, 或用二进制数表示列的顺序号。例如, (7, 4) 码的海明码 H 矩阵第一列为 001, 第二列为 010, 等等, 即

$$H_{(7,4)} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (6.3-15)$$

那么, 只要得到一个伴随式为 010, 立即就可知道接收到的码字上第二位出了差错。这样安排的 H 矩阵译码就比较简单。

海明码有 k 位信息, r 位校验位, 在 2^r 个状态中用一个状态表示正确码字, 其余 $2^r - 1$ 个状态用于判断 k+r 位中任一位出现错误, 有

$$2^r - 1 \geq k + r \quad (6.3-16)$$

称此为海明不等式。例如, (7, 4) 码中, k=4, r=3, 满足 $2^3 - 1 = 7 \geq 4 + 3$ 。可以检验由

(7, 4) 码生成矩阵产生的所有 $2^4 = 16$ 个码字, 每个码字的重量至少是 3, 所以它满足我们前面讨论的 $d_{\min} \geq 2t + 1$ 可以纠正 t 个错的结论, 当 t=1 时, $d_{\min} \geq 3$, 这种码只能纠正一位上的错误, 不能发现两位上同时出现的错误。

对能纠正单个错误的海明码, 若遇到了两位上同时发生错误时, 错误模式中有两位 (如 e_i, e_j) 为 1, 此时伴随式 S 是 H 矩阵中第 i 列与第 j 列相应位按模 2 求和结果的转置。因 H 的各列互不相同所求异或和结果肯定不为 0, 所以可判定有错误存在, 但伴随式与 i、j 列都不同, 它可能与 i、j 列以外的 k 列相同, 结果判定第 k 位出错, 这不但没把 i、j 位的错误纠正过来, 反而将正确的第 k 位的码元给纠错了。为了能纠正单错, 发现双错, 可在纠单错的海明码上再加一位偶校验位。将码的最小距离从 3 增到 4。使它满足

$$d_{\min} \geq e + t + 1$$

能纠 t 个错, 同时检测 e 个错的结论。当 t=1, e=2 时, $d_{\min} \geq 4$, 构成 (8, 4) 码。我们把这种码称为扩展海明码。具体构成法如下:

增加的一位是对所有 n 位码元的模 2 和。扩展海明码的 H 矩阵是在原海明码 H 矩阵的最右侧加上全 “0” 的列, 然后再在最下面 (或最上面) 加上一行全 “1” 的行, 最后构成的扩展海明码一致校验矩阵如图 6.19 所示。所以求任意两列之和时, 最下面一位数一定为 “0”,

$$H_{(8,4)} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} H_{(7,4)} & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

图 6.19 一种扩展海明码的一致校验矩阵

这就保证了任何两位出错的伴随式都与 H 矩阵中任何一列不同, 它只能报告发生错误, 而不致发生乱纠某一位码元的现象。因此扩展海明码能达到纠正所有单个错误, 同时能发现所有两位上的错误。

对推广的海明码, 校验位数增加 1, 设 r' 为推广海明码的校验位数, r 是原海明码的校验位数, 则 $r' = r + 1$, $r = r' - 1$ 。由式 (6.3-16) 得 $2^{r'-1} \geq k + r'$ 把 r' 换成 r , 则对纠一检二的扩展海明码讲, 如 r 为其校验位数, r 应满足不等式

$$2^{r-1} \geq k + r \quad (6.3-17)$$

在实现用于磁心存贮器的纠错编码时, 必须注意奇偶校验矩阵的合理选择。除了要满足理论上的要求外, 还必须注意工程上的要求。如要求组件较省, 尤其是级数要少, 有时还得兼顾划分插件的方便。

在这里附带指出, 由于个别磁心性能不一致及性能的变化, 或因生产测试等一系列工艺流程中偶然的机械损坏等原因可能造成个别磁心的破损。对于磁心存贮器的这一类故障, 可以在系统运行过程中用备用单元自动更换故障单元。这时对故障单元的访问就自动转变为对其备用单元的访问。因此, 从使用角度看, 并不知道故障单元的存在。故障单元就象无故障单元一样分配。这就提高了存贮器的容错能力和系统的使用效率。随着大规模集成电路半导体存贮器的出现, 故障单元自动更换措施日益受到重视。对于个别单元有缺陷的半导体存贮器, 可以在系统中通过故障单元自动更换措施当作正品来使用。这样就等于大幅度提高了大规模集成电路半导体存贮器的成品率。

3.2 循环冗余码在磁带机中的应用

循环冗余码是循环码的一种, 本节专门介绍循环冗余码在磁带记录中的应用。至于一般的循环码, 这里不作介绍。有兴趣的读者可以参看纠错码的专著。

循环码一般习惯用多项式来表示。如有一个 n 位 = 进制数字 $(V_0, V_1, \dots, V_{n-1})$, 一般表示为多项式

$$V_0 + V_1 X + V_2 X^2 + \dots + V_{n-1} X^{n-1}$$

这里, $V_i \in \{0, 1\}$, $i = 0, 1, \dots, n-1$ 。多项式中的加法和乘法分别为“逻辑异或”和“逻辑乘”。

在磁带上产生错误的原因主要是涂层的缺陷、灰尘、损伤等。这些缺陷的大小常在 1 毫米以下。1/2 吋磁带有 9 个磁道, 道与道之间的距离约为 1.3 毫米。在一个磁道上, 如果信息记录密度为 800 位/吋的话, 每 1 毫米上可有 32 位信息。因此这些缺陷往往只影响一个磁道, 而且会影响到一个磁道中的多位错。

1/2 吋磁带有 9 个磁道, 构成了一行上的 9 位。其中 8 位是信息位, 表示一个字符。另一位是这 8 位的奇偶校验位, 通常用奇校验。读出时, 将每一行上的 9 位按位加, 若结果为“1”, 则认为没有出错, 如果结果为“0”, 则认为本行在某道上的一位出错。这就是所谓的垂直冗余校验码 (VRC)。磁带带面上的信息又是按信息块记录的。若干个字符构成一个信息块。每个信息块设一个水平冗余校验字符 (LRC)。此字符也是 9 位, 每位用来校验这一道在本记录块中的奇偶性。一般是用偶校验, LRC 校验字位于记录块的末端。

显然, 用 VRC 和 LRC, 只能检测一道上的一位错, 对一道上的多位突发性错, 仅用

RC和LRC是不能检测出来的。循环冗余码(CRC码),就是在VRC和LRC的基础上,再增加一个循环冗余校验字符,用以检测哪一道出多位错。再利用VRC位在重读时进行校正。

图6.20表示的是一个记录块内带面信息的情况。信息块中有 n 个字符,每个纵行表示

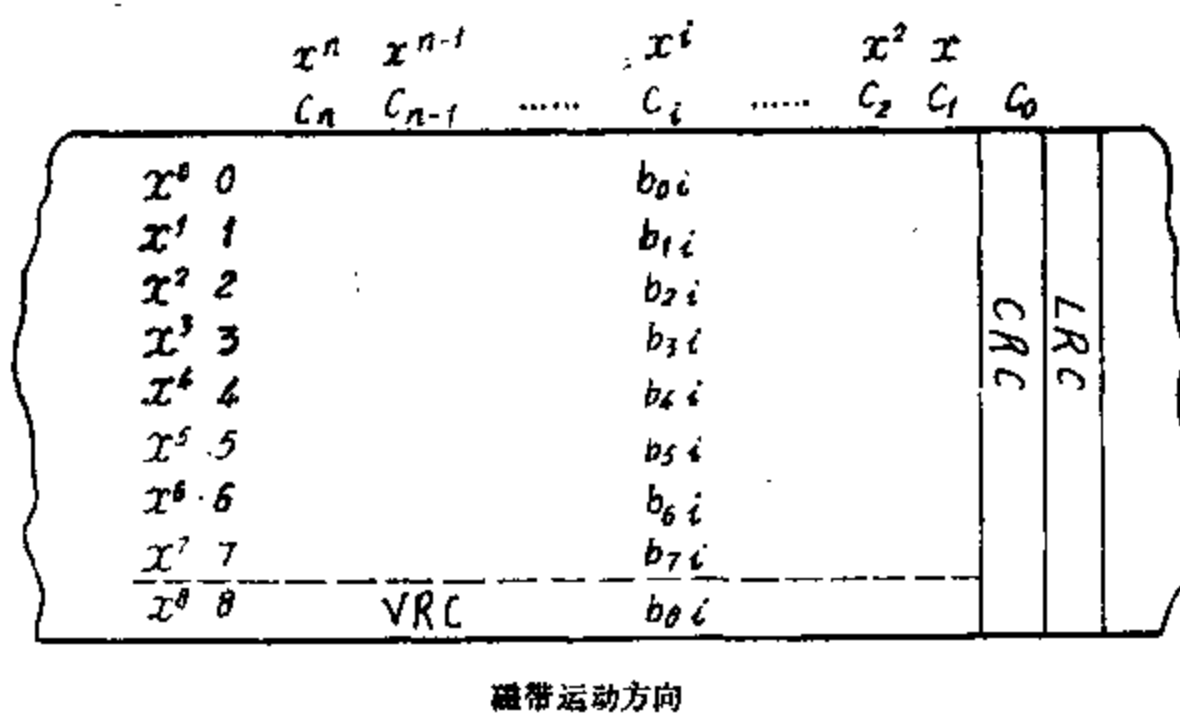


图6.20 磁带记录块内信息分布

一个字符。 C_n 表示第1个字符, C_1 表示最后一个字符。字符 C_i 由9位二进制数字组成 $b_{0i} b_{1i} \dots b_{8i}$ 。一般 C_i 用多项式表示

$$C_i = b_{0i} + b_{1i}x + \dots + b_{8i}x^8$$

现在来看一下编码过程,即如何形成CRC字符。CRC码,作为循环码,有一个生成多项式。现在这里采用的生成多项式为

$$g(x) = 1 + x^3 + x^4 + x^5 + x^6 + x^9$$

如用 C_0 表示CRC字符,它应按下列公式计算:

$$C_0 = \sum_{i=1}^n x^i \cdot C_i \text{ Mod } g(x)$$

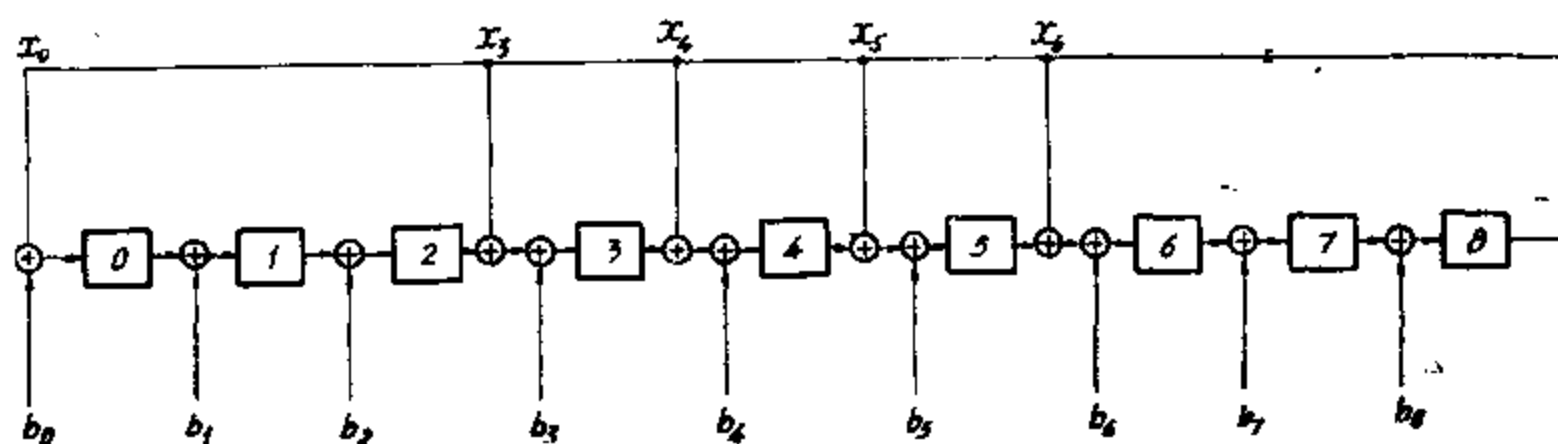


图6.21 CRC寄存器

在具体实现时,可以采用如图6.21所示的线性开关电路来计算CRC字符。图中 \oplus 表示异或门,方框表示存贮器件,它有一个输入,一个输出,其输出等于前一个单位时间的输入。从这点上看,它可以是一个延迟器件,也可以是一个单级移位寄存器(如D触发器)。图中没有画出移位寄存器同步信号。图6.21的移位寄存器称为CRC寄存器。

开始时,CRC寄存器内容为0。磁带机在写入字符 C_n 时,也把 C_n 送到CRC寄存器的9个并行输入端 b_0, b_1, \dots, b_8 ,然后移位同步信号把原CRC寄存器内容右移一位加上 C_n 再送入CRC寄存器。这时CRC寄存器的内容为 C_n 。当写入 C_{n-1} 时, C_{n-1} 也出现在9个输入端上,移位同步信号又把原CRC寄存器内容右移一位加上 C_{n-1} 送入CRC寄存

器。原来CRC寄存器里是 $C_n = b_{0n}x^0 + b_{1n}x^1 + \dots + b_{8n}x^8$ ，右移一位后形成了

$$xC_n = b_{0n}x + b_{1n}x^2 + \dots + b_{8n}x^9, \text{ 再加上}$$

$$C_{n-1} = b_{0n-1} + b_{1n-1}x + \dots + b_{8n-1}x^8 \text{ 后形成的}$$

$C_{n-1} + xC_n$ 的最高项可能是 x^9 (如果 $b_{8n} = 1$ 的话)。从图 6.21 中看出第 8 个寄存器中的 1 信号 (x^8) 右移一位后把 $1 + x^3 + x^4 + x^5 + x^6$ 反馈给 CRC 寄存器。我们知道， x^8 右移一位后应变成 x^9 ，而 $1 + x^3 + x^4 + x^5 + x^6$ 实际上是 x^9 除以 $g(x) = 1 + x^3 + x^4 + x^5 + x^6 + x^9$ 所得的余数。因此，在这时CRC寄存器中的内容是 $(C_{n-1} + xC_n) \text{Mod } g(x)$ 。这样，随着字符的逐个输入，CRC 寄存器的内容相继为

$$\begin{aligned} & C_n \\ & (C_{n-1} + xC_n) \text{Mod } g(x) \\ & (C_{n-2} + x(C_{n-1} + xC_n) \text{Mod } g(x)) \text{Mod } g(x) \\ & \quad = (C_{n-2} + xC_{n-1} + x^2C_n) \text{Mod } g(x) \\ & \quad \vdots \\ & (C_1 + xC_2 + \dots + x^{n-1}C_n) \text{Mod } g(x) \\ & \quad = \sum_{i=1}^n x^{i-1}C_i \text{Mod } g(x) \end{aligned}$$

在 C_1 输入 CRC 寄存器后，再右移一次，CRC 寄存器就得到 CRC 字符。

$$C_0 = \sum_{i=1}^n x^i C_i \text{Mod } g(x)$$

此 C_0 就作为 CRC 字符记入磁带。

这里要说一下 CRC 字符的奇偶性。 C_n, C_{n-1}, \dots, C_1 ，由写入时，VRC 奇校验位的作用，都含有奇数个“1”。 C_0 是计算出来的，它的奇偶性如何呢？首先，对于 $g(x) = 1 + x^3 + x^4 + x^5 + x^6 + x^9$ 这样含有偶数项的生成多项式，CRC 寄存器右移一次并不改变 CRC 寄存器内容的奇偶性。而开始时，CRC 寄存器为“0”，即含有偶数个“1”，输入 C_n 后，就含有奇数个“1”了，再输入 C_{n-1} 后，又含有偶数个“1”。因此，当 n 为奇数时， C_0 含奇数个“1”，当 n 为偶数时， C_0 含有偶数个“1”。

下面看看在读出时如何检测和纠错。

在读出装置中有一个同写入时一样的形成 CRC 字符的 CRC 寄存器。在读出信息块的每一个字符时，除了把它送到有关装置外，还同时送入 CRC 寄存器。CRC 寄存器原始内容也是 0。信息字符送入 CRC 寄存器的次序与写入时一样，从 C_n 开始，一直到 C_1 。这

样，如果读写过程没有出错的话，当输入 C_1 后，CRC 寄存器内容为 $\sum_{i=1}^n x^{i-1}C_i \text{Mod } g(x)$ 。

再右移一次，并输入 C_0 后，CRC 寄存器内容为

$$\left(C_0 + x \sum_{i=1}^n x^{i-1}C_i \right) \text{Mod } g(x) = 0$$

如果在第 j 道，字符 C_i 所在的行的一位二进制数字出错，即 $C_i = b_{0i} + b_{1i}x + \dots + b_{8i}x^8$ 中的 b_{ji} 变成了 $b'_{ji} = b_{ji} + 1$ ，则在读完信息块和 C_0 之后，CRC 寄存

器的内容为

$$C_0 + \left(\sum_{i=1}^n x^i C_i + x^j x^i \right) \text{Mod } g(x) \\ = (x^j x^i) \text{Mod } g(x)$$

如果在第 j 道上有多位错, 最后 CRC 寄存器的内容为

$$C_0 + \left(\sum_{i=1}^n x^i C_i + x^j E \right) \text{Mod } g(x) \\ = (x^j E) \text{Mod } g(x)$$

其中 $E = \sum x^i$, 表示 j 磁道上的错误模式, i 表示出错的字符 C_i 的序号 (也可能包括 C_0)。

在读出装置中, 除了 CRC 寄存器, 还有一个错误模式寄存器 EMR, 见图 6.22。它的构造和 CRC 寄存器一样。所不同的是它只有一个输入端。这个输入由每个字符的 VRC 校验结果提供: 若字符有奇数个“1”, 表示无错, 就输入 0; 若字符有偶数个“1”, 则有一位错, 就输入“1”, 相当于向 EMR 输入一个 x^8 。因此在读完一个信息块 (包括 CRC 字符 C_0) 之后, EMR 的内容应为 $(x^8 E) \text{Mod } g(x)$

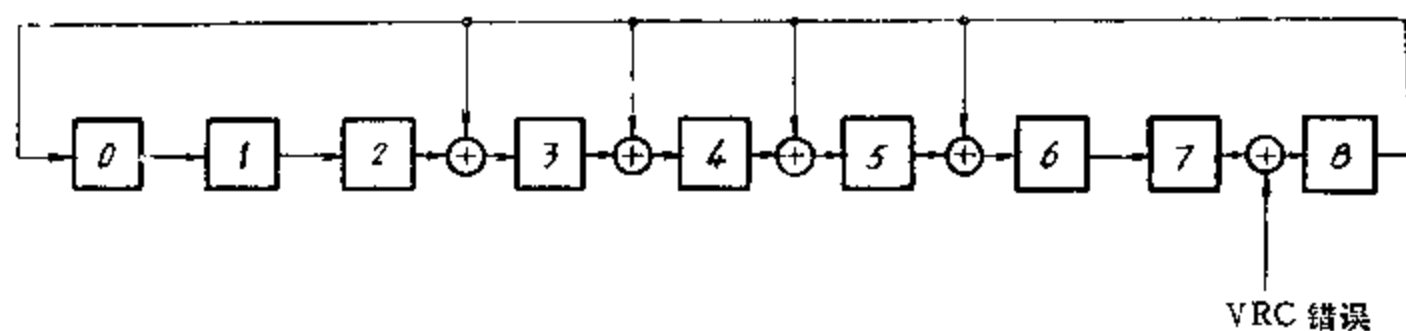


图 6.22

这时把 CRC 寄存器内容和 EMR 寄存器内容比较, 如果相等, 即

$$(x^j E) \text{Mod } g(x) = (x^8 E) \text{Mod } g(x)$$

可以得出 $j=8$, 即第 8 道出错。若二者不相等, 则将 CRC 寄存器内容右移一次 (即乘 x), 再与 EMR 内容相比较, 如相等即

$$(x^{j+1} E) \text{Mod } g(x) = (x^8 E) \text{Mod } g(x)$$

则 $j+1=8$, 得 $j=7$ 。如不等, 再如此进行下去。总之, CRC 寄存器右移 k 次 ($k \leq 8$) 之后, 与 EMR 内容相等, 则

$$j = 8 - k$$

若 CRC 寄存器右移 8 次后, 还不与 EMR 内容相同, 说明有两个以上的磁道出错。这时只能发现错, 而不能指出哪一个道上出错。

校正过程是将记录块重读, 对凡是 VRC 发现出错的字符中在出错的道上那一位进行修改。如为“1”则改为“0”; 如为“0”则改为“1”。整个记录块重读完 (包括 CRC 字符 C_0), 校正过程也就结束。

以上介绍了存储器 and 外部设备中误差校正码的应用。

在中央处理部件中, 信息的传送可以采用奇偶校验, 并以加法器和总线为中心放置奇偶

预测逻辑来检测运算中的差错。奇偶预测就是根据寄存器中的初始信息和奇偶位以及所执行的操作，预先快速地判出结果的奇偶位，并和操作结果形成的奇偶位进行比较，用以检测从全加器到母线的整个数据流程是否执行得正确。

运算器中的差错也可以用剩余码来检测。设 $r(x)$ 表示数 x 的模 a 剩余码。如果要检查 $A+B$ 是否正确执行，一方面将 $r(A)$ 、 $r(B)$ 送入检测码运算器得 $r(A)+r(B)$ 的模 a 剩余，另一方面，将 $A+B$ 送入模 a 编码，得到 $r(A+B)$ 。当两者不相符时，即

$$r(A)+r(B) \neq r(A+B) \pmod{a}$$

就说明操作出错，可以由比较器发出出错信号。剩余码既适用于运算的检查，也适用于传送的检查。如模数较小，校验线路相当简单，但速度较慢；如采用较大的模数，速度可以加快，但校验线路相应复杂化。

无论是奇偶预测，还是剩余码校验，它们仅是一种检测的手段，还必须和自修复手段相结合，才能达到度过暂时性故障，实现容错的目的。

§ 4 可靠性模型和分析

提高计算机系统可靠性的一个办法是采用适当的冗余技术。冗余可以以不同的形式和在不同的级上引入。一般来讲，冗余系统要比无冗余系统可靠性来得高，同时，冗余系统要增加元器件，增加了系统的代价。因此，如何使额外的代价小，而得到的冗余系统的可靠性又高，才是恰当的冗余容错技术的设计目标。这就需要一个能正确衡量可靠性高低的数学模型。

另外，一个计算机系统通常是由好几个子系统组成的。只有当这些子系统均正常工作时，整个系统才能正常工作。整机的可靠性就由这些子系统的可靠性决定。这里就有一个各子系统之间可靠性的平衡问题。如果有一个子系统的可靠性特别低，而别的子系统即使用各种冗余技术保护得很好，整个系统的可靠性还是不高的。有了可靠性的数学模型，可以分析出各子系统的可靠性大小，从而有的放矢地对最不可靠的子系统加强冗余保护，减少盲目性。

4.1 可靠性模型

一个系统（子系统、模块、元件）的可靠性定义为从它开始运行（ $t=0$ ）到某时刻 t 这段时间内能正常工作的概率。用 $R(t)$ 表示，显然它是时间 t 的减函数，而且有 $R(0)=1$ ， $R(\infty)=0$ 。

从另一方面看，系统从开始运行起，随时有可能失效，所以系统失效所发生的时刻是一个随机变量 ξ ，如用 $F(t)$ 表示系统在时刻 t 之前失效的概率，即

$$F(t) = P\{\xi \leq t\}$$

则在数学上， $F(t)$ 是随机变量 ξ 的概率分布函数。同样，如果要用随机变量 ξ 表示 $R(t)$ ，根据 $R(t)$ 定义，有

$$R(t) = P\{\xi > t\}$$

因此， $R(t) = 1 - F(t)$

一般，系统的失效是由元器件的失效引起的，因此系统的可靠性与元器件的可靠性直接有关。因为系统的元器件都是集成电路的模块，这些模块的失效试验统计资料，对归结系统

的可靠性模型是十分有用的。图 6.23 给出的是典型的元件失效率 $\lambda(t)$ 关于时间的曲线，失效率就是单位时间内失效的元件数与元件总数的比例（在同工艺类型的组件中），即

$$\lambda(t) = - \frac{dn(t)}{n(t)} / dt$$

$-dn(t)$ 表示失效的元件数。这里引进的时间增量 dt 是必要的。因为，如果所考虑的时间间隔是 0，则失效的元件数为 0，如所考虑的时间间隔为无穷大，则失效的元件数肯定等于元件总数。因此只有把时间因数考虑进去，才有意义。

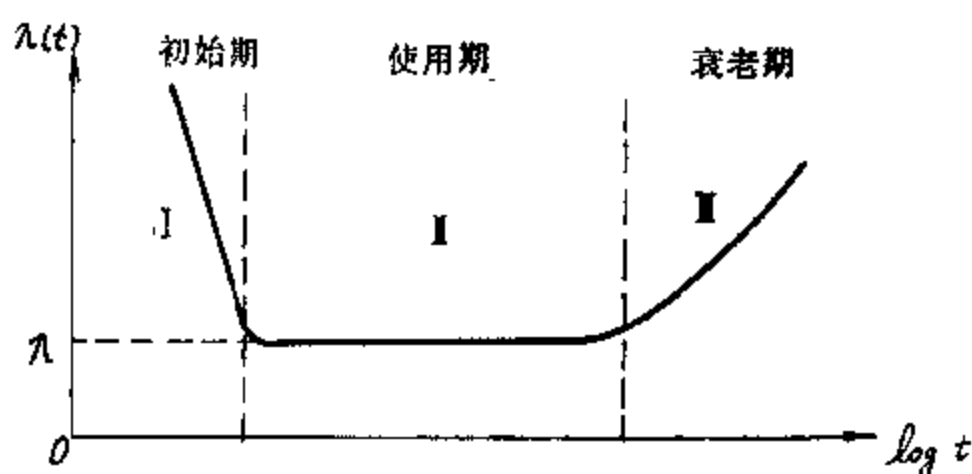


图 6.23 元件失效率曲线

对一个元件来讲， $-\frac{dn(t)}{n(t)}$ 是它失效的概率。因此 $\lambda(t)$ 也是一个元件在 0 到 t 时刻完好，而在以后的单位时间内失效的概率，即：

$$\lambda(t) = P\{\xi \leq t + dt | \xi > t\} / dt$$

图 6.23 的曲线本身很难解析地模型化。但我们注意到曲线可以粗略地分成三段。第 I 段对应的是元件的初始期，这相当于元件刚出厂后的老化筛选阶段。因性能还未达到稳定，失效率也较高。第 II 阶段为使用期，失效率基本上保持常数。第 III 阶段为衰老期，因为元件开始变质，失效率也提高了。而实际上，装入系统的元件往往是处于第 II 阶段。可以认为 $\lambda(t)$ 是一个常数 λ 。因此

$$\begin{aligned} \lambda &= - \frac{P\{\xi \leq t + dt | \xi > t\}}{dt} \\ &= - \frac{P\{\xi \leq t + dt \cap \xi > t\}}{P\{\xi > t\} dt} \\ &= \frac{P\{t < \xi \leq t + dt\}}{P\{\xi > t\} dt} \end{aligned}$$

因为 $F(t)$ 是 ξ 的概率分布函数， $F(t) = 1 - R(t)$ ，所以

$$\lambda = \frac{F(t + dt) - F(t)}{(1 - F(t)) dt} = \frac{dF(t)}{R(t) dt} = - \frac{dR(t)}{R(t) dt}$$

解此方程，并由边界条件 $R(0) = 1$ ，得

$$R(t) = e^{-\lambda t} \quad (6.4-1)$$

$$F(t) = 1 - e^{-\lambda t} \quad (6.4-2)$$

式 (6.4-1) 也可以作为系统可靠性的数学模型，其中失效率 λ 包含很多因素，如系统的复杂性，逻辑设计的弱点，组装布线和工艺上的缺点以及工作环境的严峻程度。这个模型虽然简单，但在实际分析中十分有用。

根据式 (6.4-2) 很容易得出一个常用的可靠性参数：平均无故障时间 (MTBF)，显然它就是随机变量 ξ 的数学期望。 ξ 的概率密度函数为：

$$P(t) = F'(t) = \lambda e^{-\lambda t}$$

因此,

$$\begin{aligned} \text{MTBF} &= \int_0^{\infty} tP(t)dt = \int_0^{\infty} \lambda t e^{-\lambda t} dt \\ &= \frac{1}{\lambda} \end{aligned} \quad (6.4-3)$$

平均无故障时间正好是失效率的倒数。

4.2 可靠性分析的例子

下面是几个可靠性分析的例子, 并假定故障的发生是互相独立的。

(1) 串联系统

如一个系统由 N 个子系统组成, 且只有在所有子系统都已正常工作, 系统才能正常工作, 这种系统称为串联系统。图 6.24 是它的可靠性分析图。如果子系统 i 的可靠性为 R_i , 则系统的可靠性 R 为:

$$R = R_1 \cdot R_2 \cdots R_N = \prod_{i=1}^N R_i \quad (6.4-4)$$

该子系统 i 的失效率为 λ_i , 即 $R_i = e^{-\lambda_i t}$, 则:

$$R = \prod_{i=1}^N e^{-\lambda_i t} = e^{-\lambda t} \quad (6.4-5)$$

这里, $\lambda = \lambda_1 + \lambda_2 + \cdots + \lambda_N = \sum_{i=1}^N \lambda_i$



图 6.24 串联系统可靠性分析

这就是说, 串联系统的失效率为各子系统的失效率之和。系统的平均无故障时间为:

$$\text{MTBF} = \frac{1}{\lambda} = \frac{1}{\lambda_1 + \lambda_2 + \cdots + \lambda_N} \quad (6.4-6)$$

因此, 串联系统的平均无故障时间总是小于各单元的平均无故障时间。串联的单元越多, 平均无故障时间就越小。

如果 N 个子系统都是一样的, 即 $\lambda_i = \lambda'$, 则有

$$R = e^{-N\lambda' t}$$

和

$$\text{MTBF} = \frac{1}{N\lambda'} \quad (6.4-7)$$

(2) 并联系统

如果一个系统由 N 个子系统组成,只要有一个子系统正常工作,系统就能正常工作。这种系统称为并联系统。图 6.25 是这种系统的可靠性分析图。因为只有在所有子系统失效时,整个系统才失效。所以如果子系统 i 的可靠性为 R_i 的话,整个系统的可靠性 R 为

$$R = 1 - (1 - R_1)(1 - R_2) \cdots (1 - R_N) = 1 - \prod_{i=1}^N (1 - R_i) \quad (6.4-8)$$

如果所有子系统是一样的,失效率均为 λ , 则 $R = 1 - (1 - e^{-\lambda t})^N$
(6.4-9)

因为

$$F(t) = 1 - R(t) = (1 - e^{-\lambda t})^N$$

所以平均无故障时间为:

$$MTBF = \int_0^{\infty} t F'(t) dt$$

$$= \frac{N}{\lambda} \int_0^{\infty} \lambda t (1 - e^{-\lambda t})^{N-1} e^{-\lambda t} d(\lambda t)$$

将上式被积函数中的二项式 $(1 - e^{-\lambda t})^{N-1}$ 展开, 可得:

$$\begin{aligned} MTBF &= \frac{1}{\lambda} \sum_{j=1}^N \frac{(-1)^{j-1} C_N^j}{j} \\ &= \frac{1}{\lambda} \sum_{j=1}^N \frac{1}{j} \end{aligned} \quad (6.4-10)$$

从上式可以看出,对某一个单元,引入 $N-1$ 个冗余单元组成并联系统能增加平均无故障时间。同时也应看到, N 较大,冗余单元所引起的平均无故障时间的增量也越小。

(3) 组件级冗余和系统级冗余

设系统由 n 个组件构成,我们再引进相当于系统的额外 $m-1$ 份付本的冗余。这里就有两种冗余方法:系统级的和组件级的。我们把系统的 m 个付本的并联系统称为系统级冗余。因此,它可以模型化为一个带有 m 个子系统的并联系统,而每个子系统又是一个具有 n 个组件的串联系统,如图 6.26 所示。我们把每个组件的 m 个付本的串联系统称为组件级冗余。

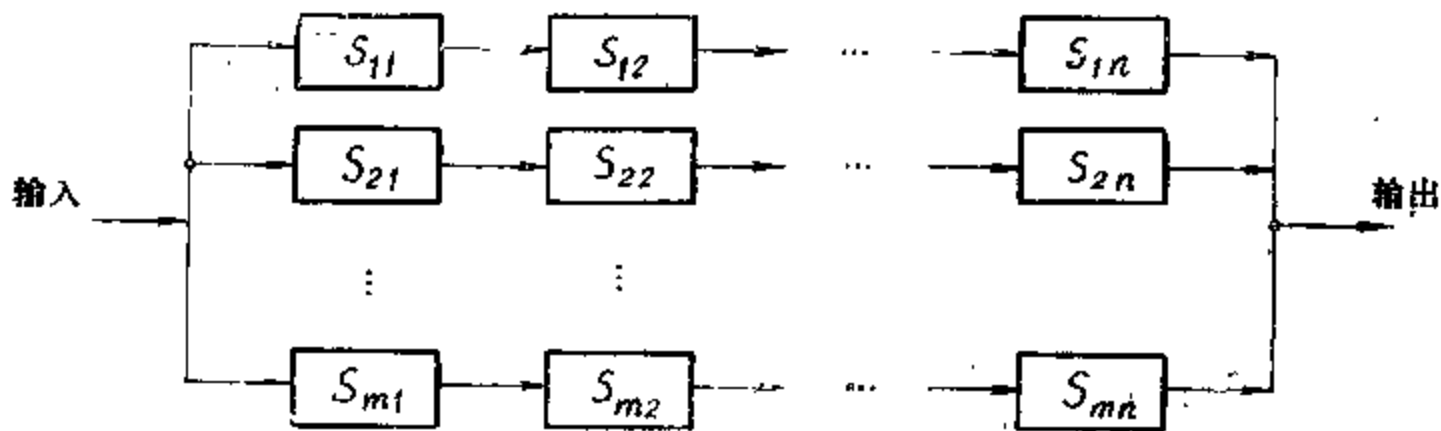


图 6.26 系统级冗余

因此它们可以模型化为由 n 个子系统组成的串联系统，而每一个子系统又是 m 个组件的并联系统，如图 6.27 所示。

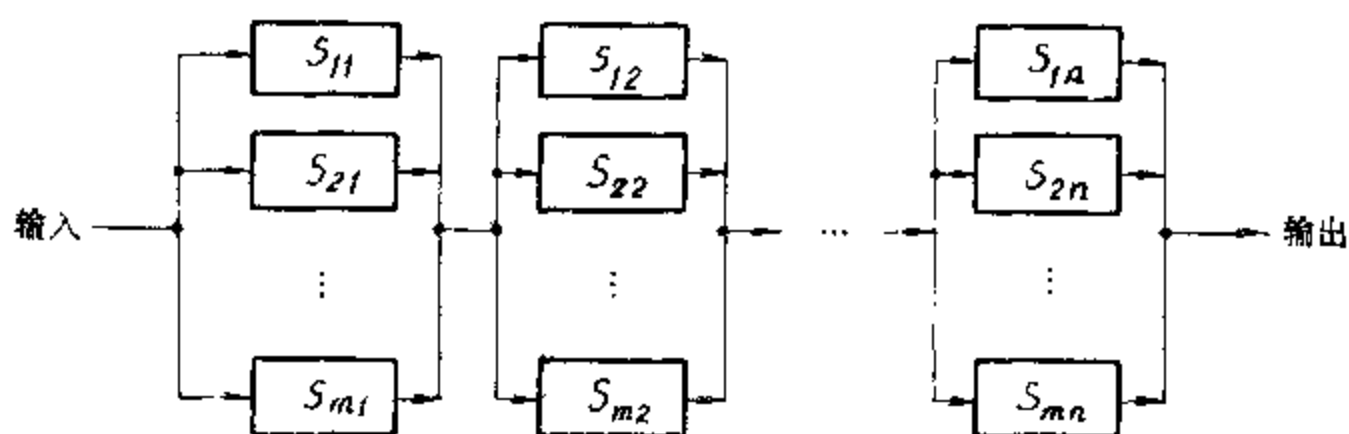


图 6.27 组件级冗余

设每个组件的可靠性为 R ，那么系统级冗余系统的可靠性 R_1 为：

$$R_1 = 1 - (1 - R^n)^m \quad (6.4-11)$$

组件级冗余系统的可靠性 R_2 为：

$$R_2 = [1 - (1 - R^m)]^n \quad (6.4-12)$$

我们可以看到 R_2 总是比 R_1 大。这就是说，组件级冗余系统的可靠性比系统级冗余系统高，但是前者需要更多的联接线，也比较难以设计。

(4) 全概率公式在可靠性分析中的应用

有些系统不能归结为简单的串联系统或并联系统，它们的可靠性分析图往往是一些比较复杂的网络，如图 6.28 所示的系统，它由 5 个子系统组成。这个系统既不是串联的 并 联系统，也不是并联的串联系统。

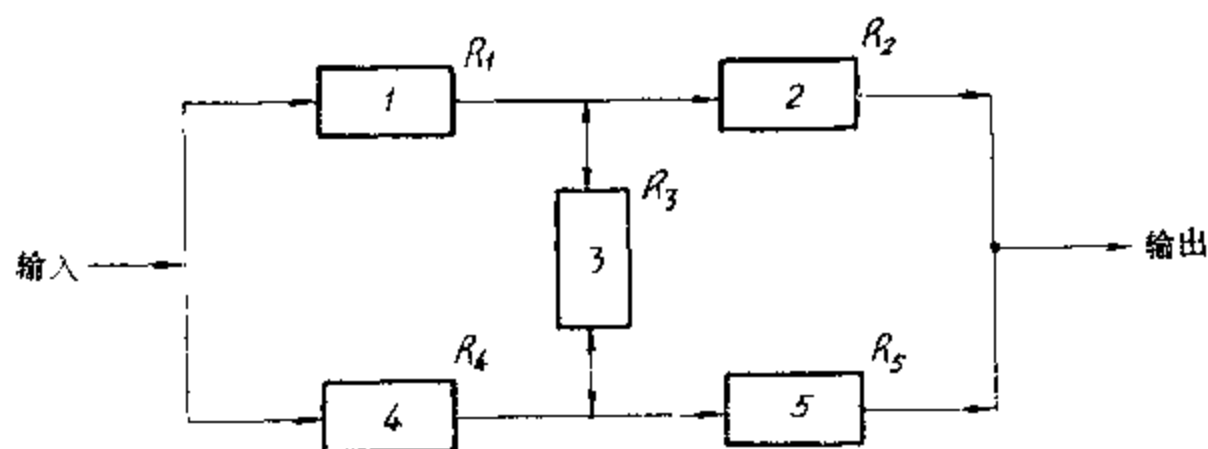


图 6.28 一个系统的可靠性分析

在这种情况下，恰当地运用全概率公式，往往能帮助算出系统的可靠性。根据概率论，如 B_1, B_2, \dots, B_n 为互不相容事件，而且 $B_1 \cup B_2 \cup \dots \cup B_n$ 为必然事件的话，则事件 A 的概率可以用下列全概率公式计算：

$$P(A) = P(A/B_1)P(B_1) + P(A/B_2)P(B_2) + \dots + P(A/B_n) \cdot P(B_n)$$

对图 6.28 的系统，我们用 S 表示“在 0 到 t 时间内整个系统正常工作”的事件，用 E 表示“从 0 到 t 时间内子系统 3 正常工作”的事件，则

$$P(S) = P(S/E) \cdot P(E) + P(S/\bar{E}) \cdot P(\bar{E})$$

这里, \bar{E} 是 E 的相反事件, $P(S)$ 就是要计算的系统的可靠性。

一般地说, 在可靠性分析图中, 如果某一子系统的可靠性为 1, 则可以用直接联接来代替它, 而与之并联的子系统全部删除, 如果某一子系统的可靠性为 0, 则可以用断路来代替它, 而与之串联在同一条支路里的子系统全部删除。

$P(S/E)$ 是在 E 事件发生下, S 事件发生的条件概率。根据上述原则, 它实际上是图 6.29 (a) 所示的系统的可靠性。它是一个并联的串联系统。因此

$$P(S/E) = [1 - (1 - R_1)(1 - R_4)] \cdot [1 - (1 - R_2)(1 - R_5)]$$

同理, $P(S/\bar{E})$ 是图 6.29 (b) 所示的系统的可靠性, 它是一个串联的并联系统。因此,

$$P(S/\bar{E}) = 1 - (1 - R_1 R_2)(1 - R_4 R_5)$$

因为 $P(E) = R_3$, $P(\bar{E}) = 1 - R_3$, 因此整个系统的可靠性为

$$R = [1 - (1 - R_1)(1 - R_4)] \cdot [1 - (1 - R_2)(1 - R_5)] R_3 + [1 - (1 - R_1 \cdot R_2)(1 - R_4 \cdot R_5)] \cdot (1 - R_3)$$

4.3 冗余结构的可靠性分析

计算机系统中可采用各种冗余结构来提高整机的可靠性。下面准备结合可靠性分析介绍三种主要的冗余结构。

4.3-1 备件替换冗余系统

在这种冗余系统中, 若干个相同的互相独立的单元同时都连到一个公共的输入上。每个单元与一个故障检测器相连。所有单元的输岀都与开关网络相连, 见图 6.30。

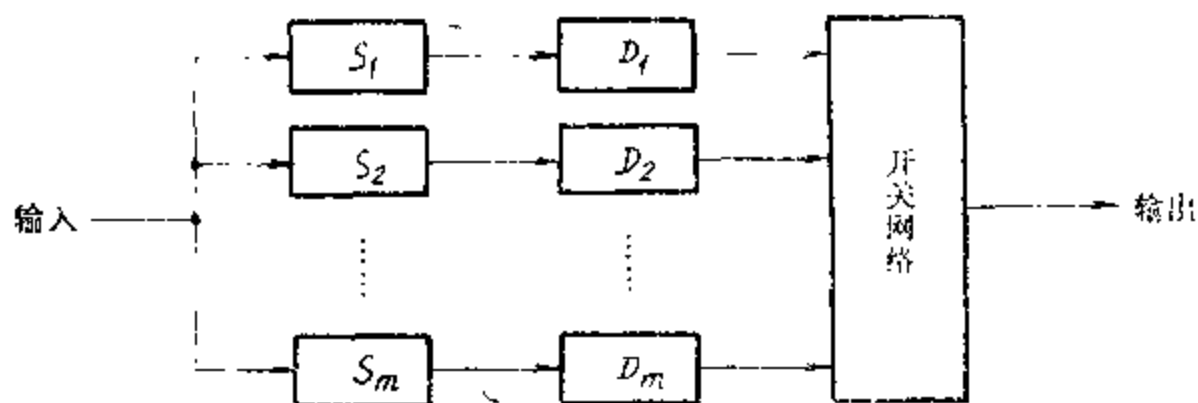


图 6.30 备件替换冗余系统

开始, 系统是把某一个单元的输出取作为系统的输出的。如相应的检测器检查出本单元有故障, 开关网络就切换到另一个正在正常工作的单元, 把该单元的输出作为系统的输出。这种过程一直继续到任务完成或者所有的单元都失效为止。在一般情况下, 单元在关掉电源时的失效率要比接上电源时的失效率要小。因此备件单元只有在开始使用时才接上电源。如果这样的话, 应先经过一些恢复过程使单元进入适当的初始状态, 以便顺利地把任务继续执行下去。

如果假设检测器和开关网络是完全可靠的 (即它们的可靠性为 1), 则只有所有单元失效时, 系统才会失效。这显然可归结为一个并联系统。如每个单元的可靠性为 R_0 , 此冗余系统的可靠性就是

$$R = 1 - (1 - R_0)^m$$

现在就可以更清楚地看出上一节提出组件级冗余系统和系统级冗余系统的区别了。虽然系统级冗余的可靠性不如组件级冗余的可靠性高, 但因为一个并联系统就有一套检测器和开关网络, 所以冗余的级别越低, 在检测器和开关网络上化的代价就越大, 线路也就越复杂。这不仅在代价上是一个沉重的负担, 而且复杂的故障检测器和开关网络反而限制了系统可靠性的提高 (实际上检测器和开关网络也不是完全可靠的)。在这种情况下, 通常在子系统级引入冗余是比较理想的。

4.3-2 N 模冗余系统

N 模冗余系统是由 N 个 ($N = 2n + 1$) 相同的逻辑线路的付本和一个表决器组成, 表决器是把 N 个付本中占多数的输出作为系统的输出, 见图 6.31。在第一节, 由基本概念中提到的 TMR 冗余系统就是 $N = 3$ 的 N 模冗余系统。

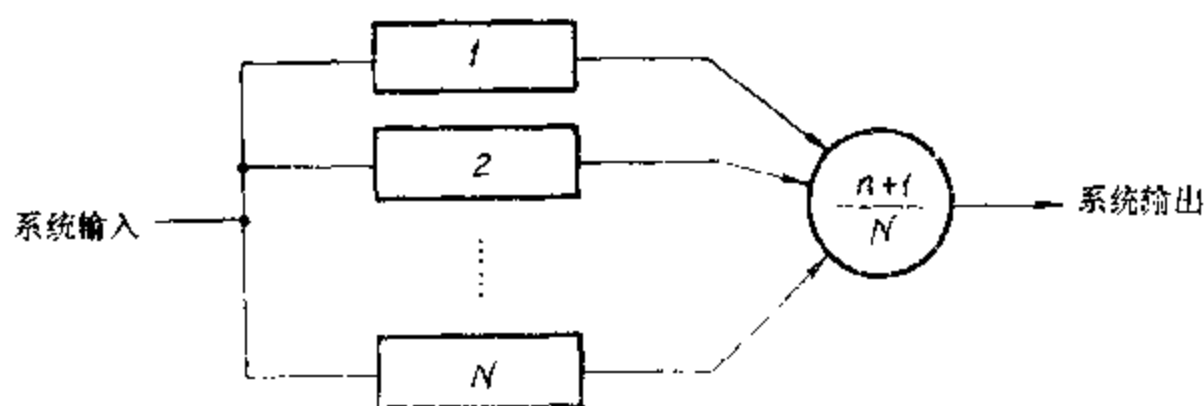


图 6.31 N 模冗余系统

显然, N 个付本中, 只要有 $n+1$ 个以上的付本正常工作, 系统就正常工作, 输出正确的结果。因此, 如果假设表决器是完全可靠的, 每个付本的可靠性为 R_0 , 则 N 模冗余系统的可靠性为

$$R = \sum_{i=n+1}^N C_N^i \cdot R_0^i (1 - R_0)^{N-i}$$

这里, C_N^i 是 N 个元素中选 i 个元素的组合数。

设 $R_0 = e^{-\lambda t}$, 则当 $\lambda t = 0.693$ 时, $R_0 = \frac{1}{2}$, 这时

$$R = \left(\frac{1}{2}\right)^N \sum_{i=0}^N C_N^i = \frac{1}{2} \cdot \left(\frac{1}{2}\right)^N \sum_{i=0}^N C_N^i$$

$$= \frac{1}{2} \left(\frac{1}{2}\right)^N \cdot 2^N = \frac{1}{2}$$

可见无论N为多少,当 $\lambda t = 0.693$ 时,可靠性都降到0.5,图6.32是N模冗余系统的可靠性曲线族。这里的可靠性曲线是相对于规范化时间 λt 而绘制的。

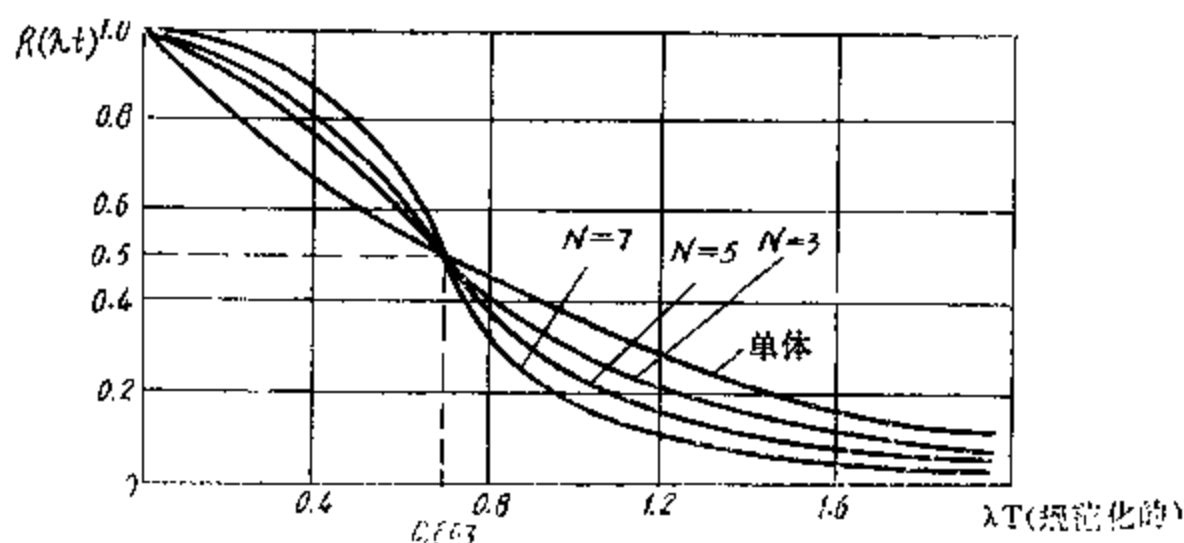


图 6.32 N 模冗余系统的可靠性曲线族

从图 6.32 可知,当 $R_0 > 0.5$ 时,引进冗余是能增加整个系统的可靠性。但如果付本本身不可靠 ($R_0 < 0.5$), N 模冗余只会使整个系统的可靠性变得更差。因为 $R_0 < 0.5$ 时,每个单体产生错误结果的可能性比产生正确结果的可能性更大,而我们又是以多数结果作为系统输出的。显然,我们使用的付本越多,就越容易输出错误结果。整个系统的可靠性就越小,若 $\lambda T = 0.693$,

$$T = 0.693 \times \frac{1}{\lambda} = 0.693 \times \text{MTBF}$$

也就是说,在单体的平均无故障时间的 0.693 之后, N 模冗余系统比单体系统更不可靠,因此, N 模冗余系统结构只适用于短时期的任务。

从图 6.32 也可看出,当 $N > 3$ 时,在可靠性方面的收益已不显著了。因此,一般取 $N = 3$ 。为了使冗余系统能长时期地可靠工作,一种可取的方法是以三模冗余为基础,再增加一些付本作为备件,形成下面要讲的混合冗余系统。

4.3-3 混合冗余结构

一个混合冗余系统 $H(N, S)$ 是以一个 N 模冗余结构作为核心和一个由 S 备件的备件库组成的。当 N 个工作的单元中有一个失效时,就用一个备件替代它,使基本的 N 模冗余结构的操作继续进行下去,见图 6.33。

混合冗余系统的物理实现由图 6.34 所示。“偏差检测器”检查 $2n+1$ 个工作的单元中是否有单元输出结果与来自恢复机构的系统输出有所不同,如发现有偏差,开关单元就把出偏差的单元切换掉,而把备件切换过来以代替这些出偏差的单元。显然,整个系统只有在 S 个备件全部耗尽,且 $2n+1$ 个基本单元中有 $n+1$ 个以上失效时才会失效。因此,如果认为

检测器、开关单元和表决器是完全可靠的，系统的可靠性可简单地表示为：

$$R = \sum_{i=n+1}^{N+S} C_{N+S}^i R_0^i (1-R_0)^{N+S-i}$$

R_0 是单元的可靠性。

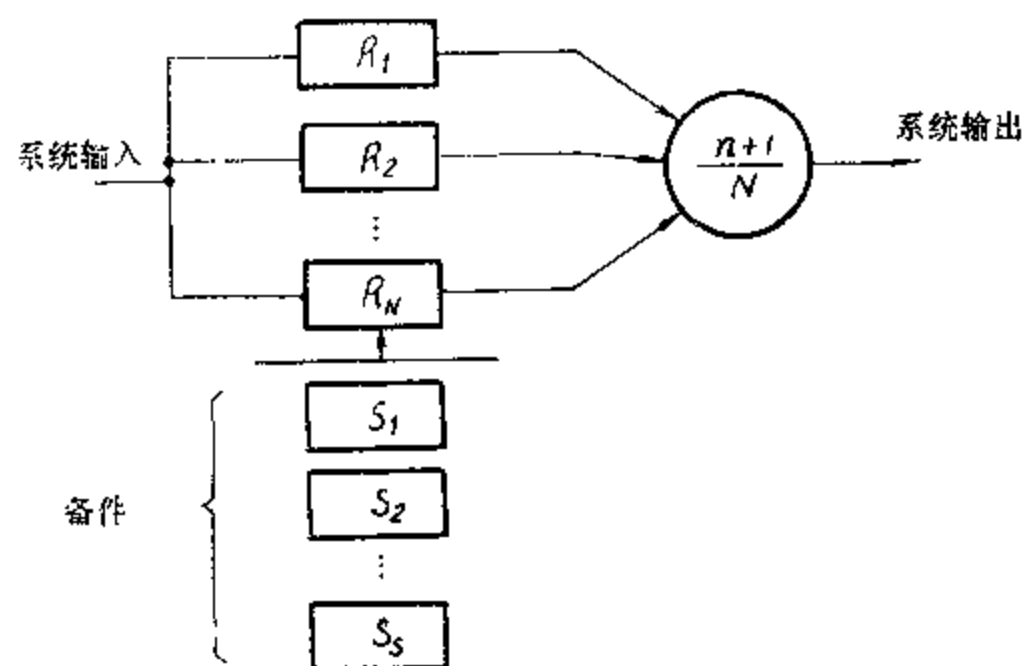


图 6.33 混合冗余结构 $H(N, S)$

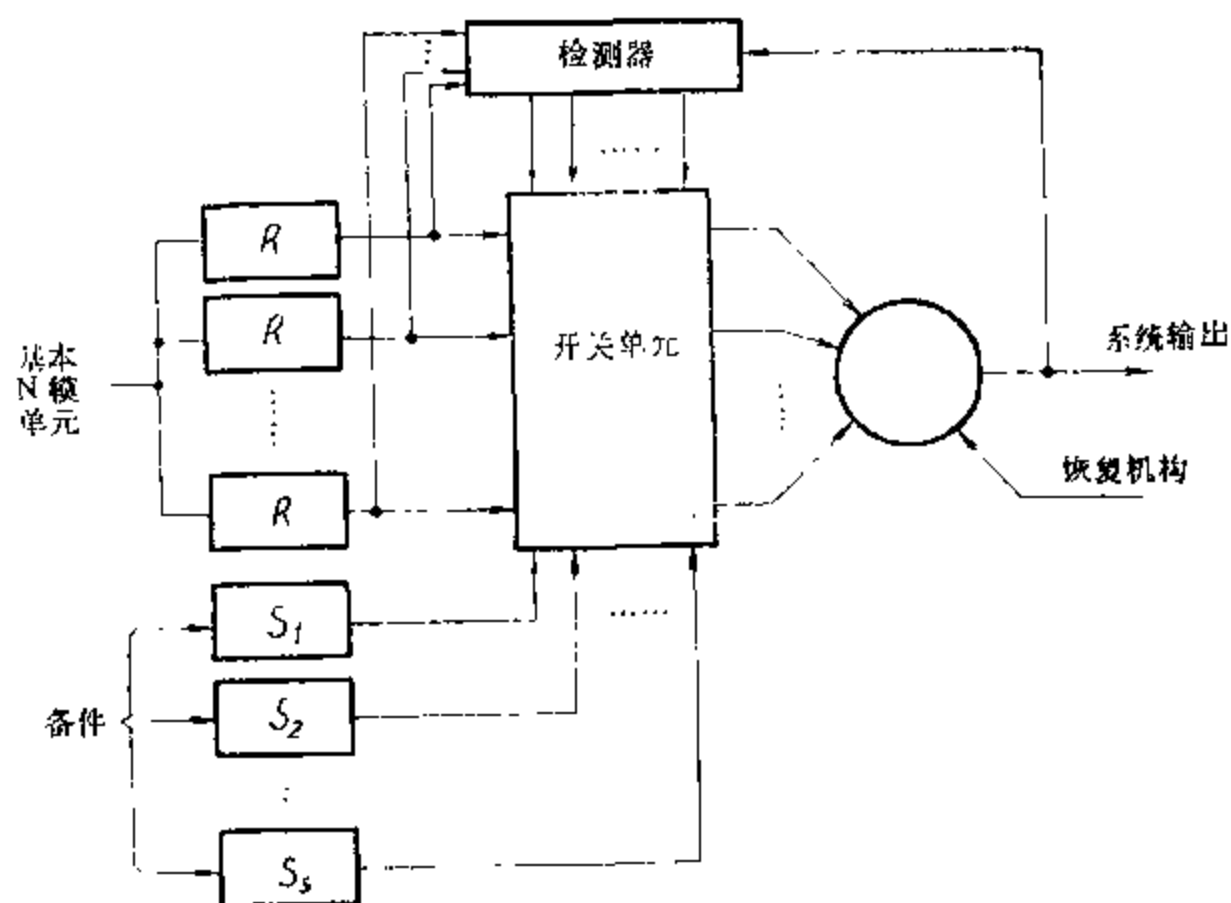


图 6.34 混合冗余系统的物理实现

另外，系统中应有适当的预防措施来禁止 $n+1$ 个以上的单元同时失效。否则，开关单元会错误地把好的单元切换掉，因为这时系统把错误的输出当作是正确的输出了。

图 6.35 给出了一个简单的 $H(3,1)$ 混合冗余系统的实例。A、B、C 是三模冗余系统的三个基本单元，D 是一个备件，表决器由逻辑门电路 $M = A_1 B_1 + B_1 C_1 + A_1 C_1$ 来实现。三个异或门 \oplus 组成了“偏差检测器”，三个触发器和一些与或门组成了“开关单元”。三个触

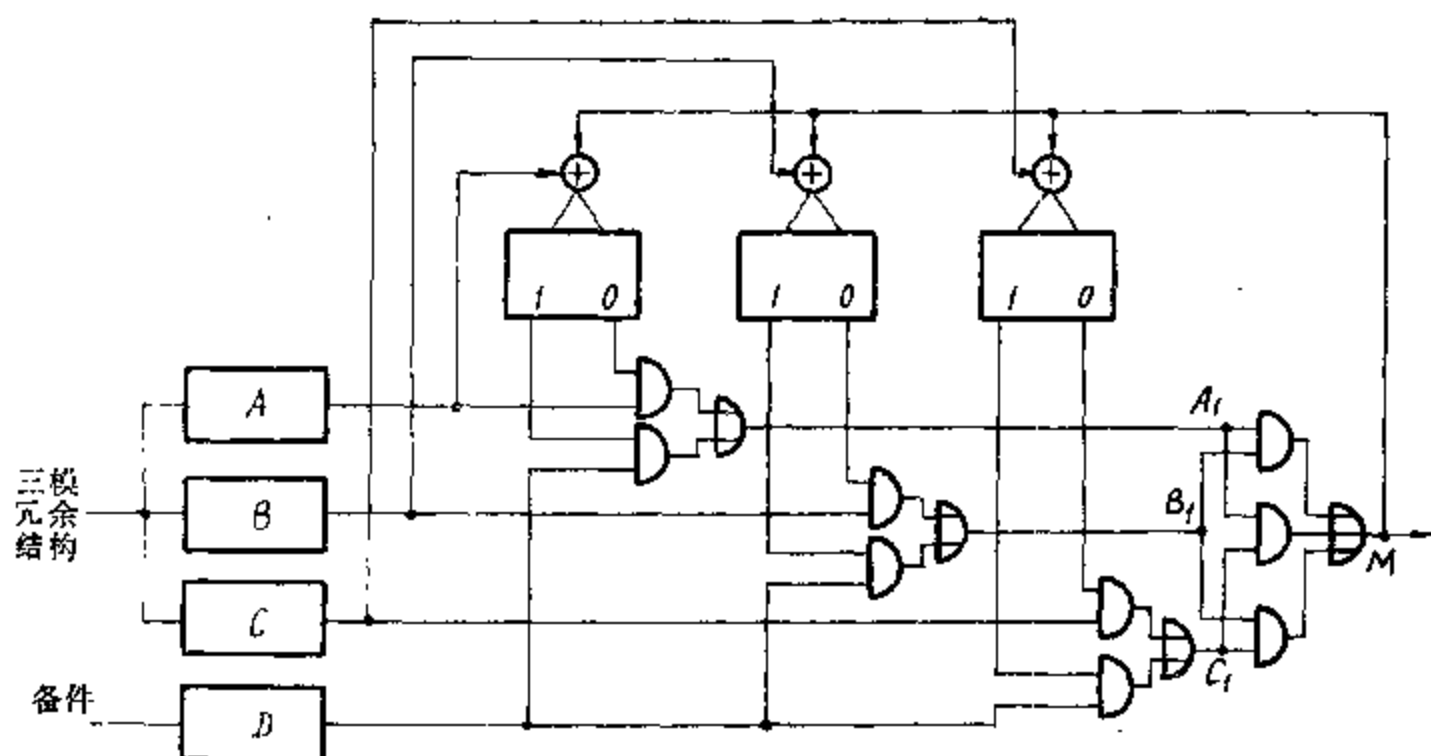


图 6.35 一个简单的 H (3,1) 混合冗余系统例

发器在开始时都置“0”。

最后，我们举一个简单的计算机系统的可靠性分析例子。设一个计算机系统有一个处理机，一个内存贮器，两个磁盘存贮器和三个磁带机。我们假设至少有一台磁盘存贮器或两台磁带机正常工作，整个系统才能完成所执行的任务。设 R_1 , R_2 , R_3 和 R_4 分别是处理机，内存贮器，磁盘存贮器和磁带机的可靠性，见图 6.36。

根据“三中取二”原则，磁带机子系统的可靠性为

$$\sum_{i=2}^3 D_i R_i (1 - R_i)^{3-i}$$

$$= R_4 + 3R_4(1 - R_4) = 3R_4 - 2R_4^2$$

因为两台盘存贮器和磁带机是并联的外存贮系统，所以，整个外存的工作可靠性为：

$$1 - (1 - R_3)(1 - R_3)[1 - 3R_4 - 2R_4^2]$$

最后得整个计算机系统的可靠性为：

$$R = R_1 R_2 [1 - (1 - R_3)^2 (1 - 3R_4 - 2R_4^2)]$$

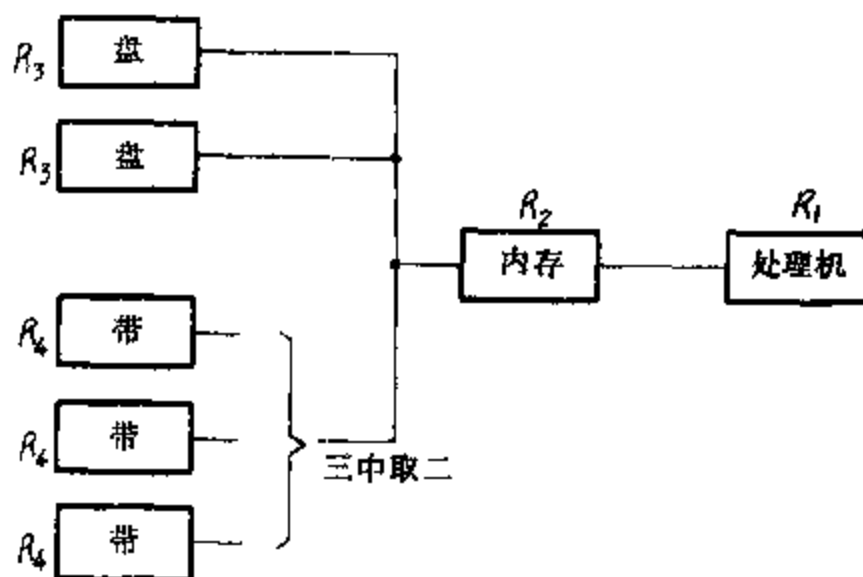


图 6.36 一个计算机系统的模型

§ 5 容错技术应用举例

本章一开始就提出，提高计算机可靠性有两条途径：提高元、器件质量和采用容错技术。当然，提高元、器件的质量是提高可靠性的主要途径。如果组成计算机的子系统所包含的小规模集成电路片数在一千以下，若组件失效率为 10^{-6} 到 10^{-7} ，则子系统的平均无故障时间理应不小于数百小时。如果达不到这个指标，就首先应该在提高元器件质量，严格老化筛选，改进工艺结构和完善逻辑设计等方面下功夫，不能寄希望于容错技术。不提高元、器件

质量,一味追求容错技术,效果会适得其反。因此,只有在元器件质量、老化筛选、工艺结构和逻辑设计等各方面都有保证的情况下,才有采用容错技术的现实意义。同样道理,双工系统也必须在单机相当可靠的基础上才值得采用。双工系统是两台同样的计算机或大体相当的计算机组成的系统。双工系统可以使两台计算机同时工作,以便互相检查,易于发现故障。也可以用一台计算机跟随另一台计算机工作,以便在一台计算机有故障时也不会使计算机工作中断。

容错的概念早在五十年代中期就已经提出,但容错计算机的出现则是六十年代后期的事。这是由于集成电路的发展导致元件价格猛跌、质量大幅度提高的缘故。在七十年代初期,硬件在计算机系统中按造价计算约占40%。目前,硬件在计算机系统中按造价计算所占比例已不到20%,而且元、器件质量也有了大幅度的提高。这就使容错技术的采用有了技术上的现实性和经济上的合理性。目前国内集成电路技术正在飞速发展之中,推广应用和加速发展容错技术已属刻不容缓。

为了具体说明容错技术的应用,就以我国的DJS-200系列中的DJS-220所用容错技术为例来介绍。在DJS-220机上,硬件冗余、软件冗余、时间冗余和信息冗余四种手段几乎都用上了,除了严格生产工艺,提高元、器件质量,提供适当的使用环境和确保逻辑设计的合理性以外,在容错技术方面所要解决的问题是:

- (1) 如何及时发现故障,防止错误扩散;
- (2) 发现故障后如何处理故障,实现容错。

发现故障的问题是用加强通路校验的办法来解决的,这是监督系统正常工作和及时发现故障的关键。DJS-220机器内部的信息传送采用按字节的奇偶校验。中央处理机内部以加法和总线为中心设置奇偶预测逻辑。所有寄存器都按字节配置奇偶位。对于某些特殊的寄存器专门设置校验线路。对于起控制作用的计数器也单独配置奇偶校验线路。对于关键性的,但其状态的变化没有规律的寄存器和触发器,例如通用状态器,则采用双份比较的方法校验。一旦通路发现故障,一般能在微指令周期内及时发现,以便为指令复执和一级中断处理提供条件,并能把出错情况和部位反映在通路校验寄存器中,供软件记录、分析和统计故障之用。

指令复执主要是利用时间冗余对付中央处理机的暂时性故障的一种容错措施。在执行一条机器指令中,如发生错误,就立即停止该指令的执行,保存好出错时的断点和当时的现场,并重取这条指令执行若干次。如不再出错,就认为复执已经成功,通过三级中断通知软件进行故障记录,原先的程序继续运行。如果指令复执一定次数后仍然出错,就认为故障的性质是固定性的,于是进一级中断处理,进行微诊断或故障定位测试,迅速查出故障,排除故障。这样也就大大提高了系统的可维修性和使用的有效性。

由于磁心存贮器的可靠性在整个系统的可靠性中占有重要的地位,因此在DJS-220机中利用信息冗余,采用了能纠正单错、发现双错的扩展海明码校验和自动更换故障单元等容错措施来提高存贮器和系统的稳定性和使用效率。

多工系统是进一步提高计算机系统可靠性的一种有效措施。DJS-220机系统结构提供了“直接读”和“直接写”指令以及通道连接上的灵活性,主存贮区前缀区的浮动等,为构成多工系统提供了方便。

早在六十年代后期,容错计算机就已经在国外出现,如美国的STAR计算机。STAR

计算机是根据动态冗余方法设计的一个替换性系统。它被分成一系列可置换的功能部件。故障检测, 恢复和置换是由专用的硬件来执行的。暂时性故障通过一段现行程序的复执来度过, 固定性故障是靠故障功能部件的置换来排除的。具体置换由电源开关实现。断开电源除去部件, 接通电源接上部件。所有部件的信息线都通过隔离电路和总线相连, 没有加上电源的部件仅产生逻辑“0”输出。作为硬核的测试修复处理机是通过带有待命贮备的三模冗余方法来得到严格保护的。在硬核部分采用静态冗余方法可以在不必进行故障诊断的情况下度过其本身的暂时性故障和一定程度的固定性故障。而且局部采用静态冗余方法对整个系统的造价影响也不大。

其它如美国的 IBM370 计算机也采用了许多可靠性、可用性和可维修性的 RAS 措施, 以进一步增加硬件的可靠性, 改善系统硬件的可维修性, 从而提高了计算机系统的使用效率。

根据国内外计算机系统的运行统计资料, 目前硬件的平均无故障时间已几倍于软件的平均无故障时间。因此, 可靠性的概念必须把软件可靠性包括在内, 而且应该比硬件可靠性占有更重要的地位。如今软件已是整个计算机系统中最庞大、最复杂、最昂贵的部分, 而且还没有理论上严格的方法能够证明某个程序是正确的。因此, 软件又是整个计算机系统中最不“保险”的部份。看来不可能有一个程序证明过程可以作为一个通用的判定过程来使用, 也就是说, 不可能有一个“万灵”的程序可以用来确定任何程序的正确性。国外已十分强调软件的工程化以及理论和实践相结合的重要性, 以提高软件的可靠性、可用性和可维修性。容错的概念必须全面地包括硬、软两个方面, 而且后者尤为重要。这不但反映了软件可靠性较低的现状, 还由于软件可以是更加灵活的强有力的容错手段。它和必要的硬件相结合, 可以根据系统出现的故障, 动态地重新组织系统的工作, 排除系统故障所引起的后果, 并在完全恢复或稍微降低系统正常功能的基础上照常工作。

目前新的容错计算机是名符其实的多机系统。平时, 系统可以全部投入工作, 但仍拥有丰富的后备资源, 也就是说, 新的容错计算机的冗余度是建立在系统结构级或功能级上的。这也是大规模集成电路的发展促使硬件价格下降所导致的必然趋势。这方面比较先进的例子有美国研制的下一代宇宙飞船用的容错计算机。它依靠软件恢复程序替换故障模块, 据称能够保证 5~10 年的自主工作。

故障保险 (Fail Safe) 是近年来很受重视的一个方案, 所谓故障保险, 指的是不管机器或部件出了什么故障, 保证不执行“灾难性”的操作 (如导弹的发射) 或执行必然是“安全性”的操作 (如电子计算机医护系统中一旦出了故障, 则不管是否需要保证把医生叫来); 又如, 对不少控制用机器, 在出故障时一般输出“0”要比输出“1”保险, 那我们就安排成在出故障时肯定只能输出“0”。为了实现故障保险, 需要在软件和硬件上都采取相应的措施, 而最主要的是要求软、硬件设计人员重视这点。

主要参考文献

- [1] M.A. Breuer and A.D. Friedman, "Diagnosis and Reliable Design of Digital Systems," Computer Science Press, 1976.
- [2] C. V. Ramamoorthy and R. C. Cheng, "Design of Fault Tolerant Computing Systems," Applied Computation Theory; Analysis, Design, Modeling, ed. T. Y.

Raymond, Prentice-Hall, 1978.

- [3] W. W. Peterson and E. J. Weldon Jr., "Error-Correcting Codes," Second Edition, MIT Press, 1970.
- [4] 南京工学院 821 教研组, "纠错码引论"。
- [5] B. Randel, "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, 1975, pp. 220-230.
- [6] C. V. Ramamoorthy and L. C. Chang, "System Modeling and Testing Procedures for Microdiagnostics," IEEE Trans. on Computers, Vol. C-21, NO. 11, Nov. 1972, pp. 1169-1183.
- [7] A. Avizienis, et al, "The STAR (Self-Testing and Repairing) Computer; An Investigation of the Theory and Practice of Fault-Tolerant Computer Design, " IEEE Trans. on Computers, Vol. C-20, No. 11, Nov. 1971, pp. 1312-1321.

第七章 多机系统

本章首先论述在计算机系统结构中引入并行性所根据的三个基本概念：时间重迭、资源重复和资源共享，同时分析实现这些概念的多机系统类型和发展趋势。然后集中介绍两种类型的多机系统：以单指令流多数据流为特征的并行处理机系统和以多指令流多数据流为特征的多处理机系统。

§1 计算机系统结构中并行性的发展和多机系统类型

本节首先指出并行性概念乃是推动计算机系统结构发展的重要因素，遵循的技术途径就是时间重迭、资源重复和资源共享，它们导致各种类型多机系统的产生和发展。因此，用并行性等级对多机系统进行分类是合理的，能取得与流行的弗林（M.J.Flynn）分类法^[1]相一致的结果。

1.1 计算机系统结构中并行性概念的发展

1.1-1 并行性的广义理解

计算机性能指标的提高，总是伴随着系统结构的新发展，它与电子元件和器件的变革是相辅相成、互相促进的关系，这是由计算机的发展历史所证明了的事实。拿最浅显的例子，即 n 位串行运算和 n 位并行运算来作比较，在元件速度相同的情况下，后者的速度几乎提高为前者的 n 倍，这就是运用最起码的并行性提高运算速度的结果。

但是，并行性又不止于设备的简单重复，它还有更多的含义。在单处理机内采用流水线指令部件和流水线运算部件，以及实现多道程序的共同执行等，这些都是最广泛意义上的并行措施。在同一时刻或是在同一时间间隔内完成两种或两种以上性质相同或不不同的工作，只要时间上互相重迭，就都存在并行性。可见，并行性（Parallelism）应该包含三重意义，即时间重迭（Time-interleaving）、资源重复（Resource-replication）和资源共享（Resource-sharing）。

时间重迭就是在并行性概念中引入时间因素，即多个处理过程在时间上相互错开，轮流重迭地使用同一套硬件设备的各个部分，以加快硬件周转而赢得速度，原则上它不要求重复的硬件设备，能保证计算机系统具有较高的性能价格比。这种并行性概念的实现在高性能单处理机中就表现为各种流水线部件和流水线处理机^[1]。

资源重复本来是根据“以数量取胜”的原则大幅度提高计算机处理速度的最直接措施，但长期以来受到硬件昂贵因素的限制而不可能普遍采用。最早的多机系统实际上并不是为了速度，而只是为了可靠性，例如 1958 年美国 SAGE 防空体系中的 IBM AN/FSQ-32 双机系统^[2]。即使在一些以提高处理速度为目标的多机系统中，处理机数目也是不多的。这里除了硬件昂贵以外，明显的障碍来自软件复杂性这一主要因素。

发展分时系统也是克服硬件昂贵的障碍、提高计算机设备使用率的另一重要措施。它所实现的是资源共享这一并行性概念，即多个用户按一定时间顺序轮流使用同一套硬件设备。即使只包含快速中央处理器与慢速外围设备在工作时间上的重迭，也可视为并行性的一种形式。更何况按一定的时间片顺序共享中央处理器，就实现了并行性更强的分时系统。它对软件中并发性的发展起着很大的作用。同时，资源共享又是计算机网和分布处理系统发展的主要动力。

因此，广义说来，并行性既包含同时性(Simultaneity)，又包含并发性(Concurrency)。前者是指两个或多个事件在同一时刻发生，后者是指两个或多个事件在同一时间间隔内发生。并行性贯穿于信息加工的各个步骤和阶段，采取着多种不同的形式，而其高级阶段就形成并行处理的专门领域。从计算机演变的历史来看，也可以认为：计算机性能的提高，除了依赖电子元件和器件的变革作为基础之外，从系统结构上引入不同等级的并行性，确实起了主要的作用。这样来理解并行性，就易于把多机系统与整个计算机系统结构的发展联系起来，作为充分发挥算法和程序并行性的特殊结构型式来加以研究。这种有机联系和发展过程将在下一节中具体讨论。

1.1-2 计算机系统结构向并行处理系统发展的途径和趋势

把上述并行性的三个方面（即时间重迭、资源重复和资源共享）综合起来，给了我们一点启示，即并行处理并不是凭空产生的，而是反映了计算机系统结构向高性能发展的自然趋势。一方面在单处理机内广泛采取多种并行性措施，另一方面由最初的独立外围计算机开始，发展各种不同耦合度的多计算机系统；从两个方面共同形成了并行处理的专门领域。这就是计算机系统结构从高性能单处理机和多计算机两极出发向并行处理发展的总趋势。

从单处理机和多计算机两极向现代并行处理系统发展的过程示意地表示在图 7.1 中。适应时间重迭、资源重复、资源共享三个并行性概念，这个发展遵循着不同的技术途径，达到不同类型的并行处理系统。

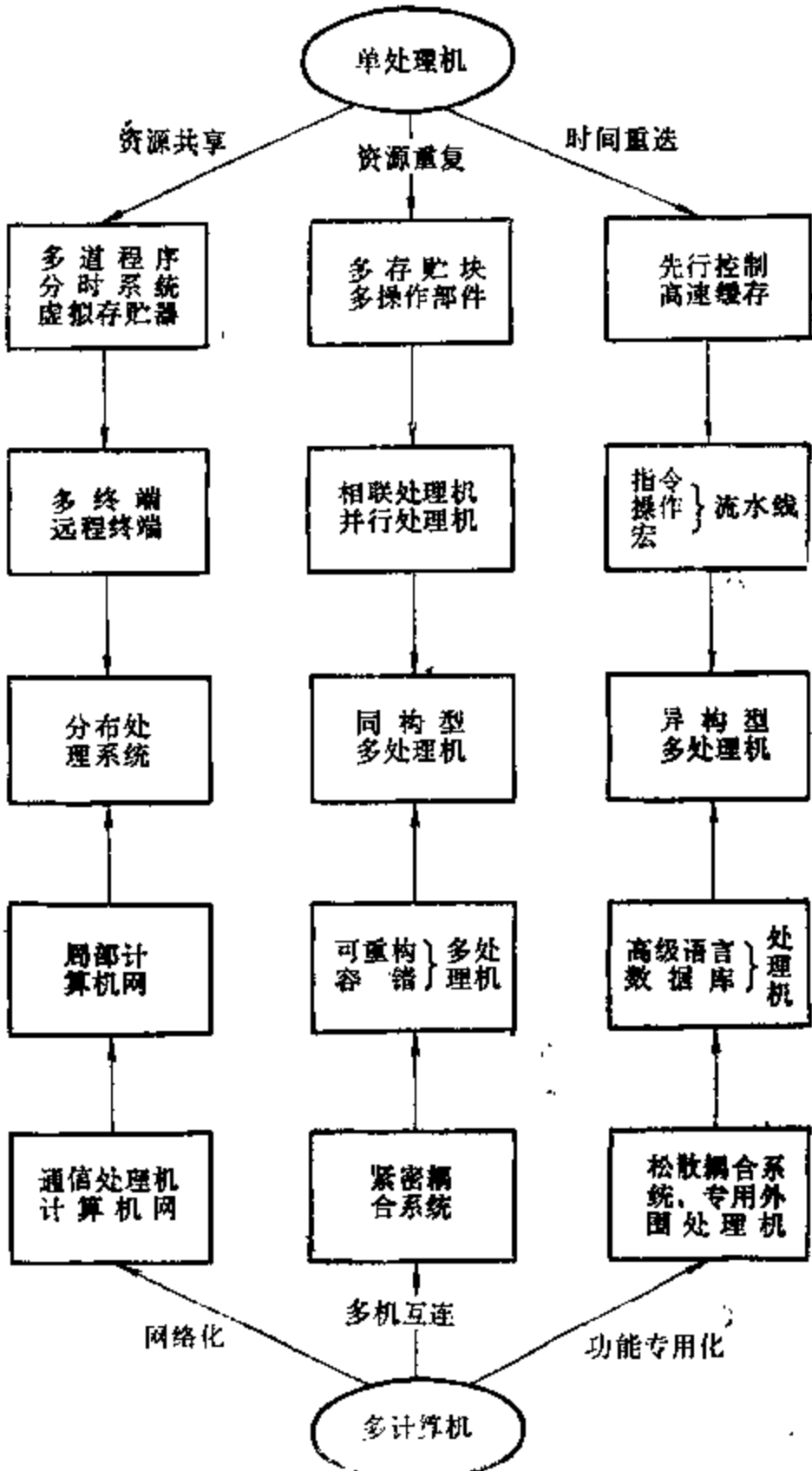


图 7.1 从单处理机和多计算机两极向并行处理发展的趋势

在高性能单处理机的发展中,起着主导作用的是时间重迭这个概念,它的基础是功能专用化(Functional Specialization),就是说把一件工作按功能分割为若干相互联系的部分,把每一部分指定给专门的部件完成,然后按流水线原则把各部分的执行过程在时间上重迭起来,使所有部件依次地分工完成一组同样的工作。高性能单处理机的发展,概括起来,可以说是不断进行功能部件的分离和谋求它们之间频带的平衡,特别是克服信息流通中的“瓶颈”,发展出高并行度的系统³。例如,为了减轻慢速输入/输出设备对快速中央处理器的过分负担,发展出输入/输出通道,直至I/O专用处理机;为了取得主存和中央处理器之间的速度匹配,采用了指令重迭、先行控制、指令和数据缓冲寄存器,直至统一的高速缓存等。按此途径发展到专门的流水线处理机,就进入了并行处理的领域。这种流水线结构可以在指令步骤和操作步骤上实现,分别构成指令流水线和操作流水线;还可以进一步发展到处理机一级,形成以任务重迭为特征的宏流水线(Macro-Pipeline)。例如,把语言编译过程分为扫描、分析、生成等部分,分别设立专门的处理机,与执行机器语言的通用处理机相连,这也是具有流水线结构的高级语言处理机的一种形式。单处理机系统中的许多辅助性或专用性功能都是可以分散的,例如系统的测试诊断、终端信息的预处理、数组运算、数据库管理等,都可以分别设置专用处理机,作为系统的选配件与主机相连,有效地提高整个系统的工作效率。这类处理机统称为非对称型(Asymmetrical)或异构型多处理机系统(Heterogeneous Multiprocessor System)。它们由多个不同类型、至少担负不同功能的处理机组成,按照程序要求的顺序,利用时间重迭原理,依次对它们的多个进程进行加工,各自完成规定的功能动作。

目前几乎所有的大型计算机系统都不只包含一台计算机,它们都不是真正的单处理机,而是某种异构型多机综合系统。至于专门研制的高效能数组处理机、数据库处理机,以及将来可能出现的专门处理高级语言编译工作等的机器,也都只有与担负其它功能的各种处理机相配合,组成统一的异构型多机系统,才能充分发挥它们各自的威力。例如,本章第二节将要介绍的BSP科学处理机¹⁸就是由专施数组运算、标量运算和系统管理等功能的三台处理机共同组成的系统。

在高性能单处理机中,资源重复概念的运用也很普遍。不论在非流水线处理机(如CDC-6600),还是在流水线处理机(如CRAY-I)中,多操作部件和多体存贮器都是得到成功应用的结构形式。这种趋势进一步发展,产生了相联处理机^[10]、并行处理机^[4]等多种按单指令流多数据流方式工作的多机系统,它们包含的多个处理单元根据单一指令流同时地对多组数据进行处理,具有很大的速度潜力。多机系统的机间互连网络在保证系统的快速性、灵活性和可靠性上起着很大的作用,发展出各种紧密耦合系统,为能动地改变自身结构的可重构多处理机、具有极高可靠性的容错多处理机等提供了理想的结构形式。这类系统统称为对称型(Symmetrical)或同构型多处理机系统(Homogeneous multiprocessor system)。它们由多个同类型、至少同等功能的处理机组成,同时地处理同一程序中能并行执行的多个任务。

同构型多处理机系统包含功能相同的多台计算机,使它们之间具备相互替换的条件,便有可能把系统的工作可靠性建立在处理机一级冗余的基础上,构成理想的容错多处理机。但是,要真正实现容错,还必须发展一种可变结构系统或可重构系统。在这种系统中,平时几

台机器都正常工作，象通常的多处理机一样；但到故障阶段就要使系统重新组织，以略为降低的规格继续运行，直到故障排除为止。当然，这种把系统容错建立在系统结构一级的做法，对机间互连网络的灵活性和可重构性又提出了进一步的要求。

对称系统的另一好处是它有很好的模块性，便于用大规模/超大规模集成电路实现，也赋予系统扩展规模和改变结构的能力。这些特点使同构型多处理机成为目前在程序级并行处理中压倒一切的结构形成，它通常以极大地提高处理速度为目标，一般都是规模很大的高速系统。

利用资源共享概念发展多道程序和分时系统也是促使单处理机向多处理机发展的另一重要途径。在原理上它们可看成是用单处理机模拟多处理机的功能，形成了虚拟存贮器、虚拟处理机等概念。近年来，由于远程终端、计算机网和微型、小型机的发展，用真实的处理机代替虚拟处理机构成以分散为特征的多计算机系统，代替以集中为特征的分时系统，已成为重要的发展趋势。本章定义为这种把大量分散、重复的处理资源（一般是具有独立功能的单处理机）相互连接起来，在操作系统（也可以是分布式）的全盘控制作用下，统一协调地工作，而最少依赖集中的程序、数据或硬件的系统称为分布处理系统^[12]。从系统结构和资源管理方面，分布处理系统为并行处理提供了可以借鉴的很好形式，它的发展使计算机网与并行处理系统之间出现相互接近的趋势。以近距离、宽频带、快响应为特点的局部计算机网（Local computer network）^[13]就是一个很好的例子。由于这种网络的局部性（由单一机关所有），机间距离不超过10公里，所以有可能使信息传输速度提高到每秒 $10^5 \sim 10^7$ 二进位，取数时间最长几十毫秒，比传统的计算机网提高一至二个数量级。如果按此方向进一步发展，将有可能满足多任务并行处理的要求。

1.2 多机系统的特性

1.2-1 多机系统的耦合度

多计算机系统中，并行性的发展主要反映在耦合度的提高上，它取决于系统的结构和计算机之间的通信能力。根据这个特征，可以将多计算机系统分为最低耦合、松散耦合、紧密耦合系统等几类。

离线(或脱机)处理系统是耦合度最低的系统，除通过某种中间存贮介质以外，计算机之间别无物理连接，也无共享的在线(或联机)硬件资源。例如独立外围计算机系统由主机和外围计算机组成，后者离线(或脱机)工作，只通过磁带、软磁盘或穿孔纸带等对主机的输入/输出提供支持。

如果计算机之间经过通道相互连接，共享一些外围设备和以较低频带在数组一级相互作用，则形成松散耦合系统(Loosely-Coupled System)或间接耦合系统(Indirectly-Coupled System)。例如大的计算中心通过开关网络把多台计算机和大量外围设备连接起来，达到相互通信和共享设备的目的，以及在更大范围内构成的计算机网都属于松散耦合系统。这种系统的特点，一般说，是其各成员的非对称性（由不同类型计算机组成）和异步工作，不同机器之间实现功能上的分工，相互配合，在共享资源和各尽所长的基础上求得较高的系统使用效率。各计算机之间经过通信机构相连，信息路径可受控制，结构比较灵活，系统规模较易扩展。但由于要花费一定的操作开销，系统的信息传输频带较窄，故难于满足任务一级并行处

理的要求。在目前系统耦合度不断提高的情况下,仍可能适合分布处理、多道程序等运行方式。

如果提高计算机之间物理连接的频带,进而使它们共享内存,则虽然相互作用仍可能在数组一级,但已具备任务级或作业级并行的条件,这样组成的系统就成为紧密耦合系统(Tightly-Coupled System)或直接耦合系统(Directly-Coupled System)。早期的直接耦合系统如IBM709 \times /704 \times 系统和由IBM360/50(或40)与IBM360/65(或50)组成的附属辅助处理机ASP(Attached Support Processor)系统³等仍都是非对称系统,按主/辅机方式配合工作,相互利用中断和通道方式在内存之间直接传送信息。辅机除了分担主机的输入/输出负载以外,还可为主机的作业进行前处理和后处理等辅助性工作,以提高系统使用效率。但是,属于紧密耦合系统的最重要类型是对称型多处理机系统,它们把多台同型号的处理机经过总线或高速开关相连,具有较高的信息传输频带和吞吐能力。这样获得的快速性和对称性都是在任务一级实现并行处理的必要条件。利用统一的操作系统管理,可保持各处理机的高效率和负载平衡。

1.2-2 多处理机的特性和优点

P·H·恩斯洛(Enslow)对多处理机提出了如下的定义:

- (1) 包含两个或多个功能大致相当的处理器。
- (2) 所有处理器都共享一个公共的内存。
- (3) 所有处理器都共享I/O通道、控制器和外围设备。

(4) 整个系统由统一的操作系统控制,在处理器和程序之间实现作业、任务、程序段、数组和数组元素各级的全面并行。

这个定义比较适用于同构型多处理机,这反映在定义的第一条关于“功能大致相当”的规定。如果把这一条理解为:功能大致相当,但不一定要求各处理器担负性质完全相同的工作,则也可适用于异构型多处理机。定义的第二条关于共享公共内存的规定比较适用于集中式结构的多处理机,其中多个处理器经过互连网络共享公共内存的多个存贮模块。如果将此条件推广到具有分布存贮器结构的多处理机,则应理解为:存贮器各个模块既可以在物理上集中,构成统一的公共内存,也可以在物理上分散属于各自的处理器,但它们之间必须经过通信网络或经过虚拟存贮器地址映象机构建立相互的逻辑连接。对于定义的第三条也可作类似的广义理解。这样一来,就把用得愈来愈多的分布处理系统引进了多处理机的范围,而定义中的处理器就成为包含内存、控制器、I/O通道在内的独立而相互联系的处理机了。

定义的第四条关于统一操作系统和全面并行性的要求是多处理机中起决定作用的因素,其它几条都是受它制约的。既然要求系统实现全面并行,那么多个处理器共享资源,特别是共享内存,就是绝对必要的。统一操作系统也是系统实现整体控制和任务调度的必要条件。所以说,全面并行性是多处理机系统最根本的特征。它把只在操作一级实现并行性的并行处理机排斥在上述定义范围之外。

由于多处理机具有上述重要特性,所以它有以下明显的优点:

- (1) 很高的性能价格比

“Computer Review”期刊曾经对1975~1977年出产的计算机性能和价格进行了统计,得到如图7.2所示的曲线^[24],证明单处理机的性能价格比随其规模增大呈现下降趋

势。但是，对于多处理机系统，如果暂不考虑机间连接等附加成本，且假定性能的提高能与处理机数目成正比的话，那么应该得到一条理想的水平直线，表示性能价格比与系统规模无关。这在实际上当然是办不到的，随处理机数目增加，性能价格比仍要变坏，但由于微小型机生产本和维护费用都大为节省，该曲线下降幅度应该会小得多。

(2) 很高的可靠性

系统包含大量同类型或同功能的计算机，使它具有很高的冗余度，能在系统结构一级重新组织，以适应容错的需要。硬性故障的排除和系统的维护也会方便得多，这样就同时提高了系统的可维护性和可用性。例如，美国为 ARPA 计算机网设计的第二代接口信息处理机 (IMP) 是一个型号为 Pluribus 的多处理机系统，最多可包含 14 台处理机，工作两年来 8 个系统的可用率平均达到 99.7%，而耗费在查错上的处理能力只占总能力的 1%。

(3) 很高的处理速度

目前单处理机加快运算速度的主要手段是提高时钟频率。例如，著名的 CRAY-I 流水线处理机的主频已经做到 80 MC。但是，这种提高终究是有限的，最终将以光速 C 作为极限。而目前实际中确又存在许多大的题目，它们对运算速度的要求是传统结构的单处理机所不能满足的，因此，有必要从多处理机方向上寻找突破口。例如，美国遥感卫星 Landsat-D 每秒平均送回地面的象素约为一百万个，每个象素的处理需经 100~10000 次运算，这就是说，单是对这一个卫星的全部信息进行实时处理，就需用每秒能完成 1 至 100 亿次运算的计算机。为此目的而设计的 MPP 多处理机系统^[16]包含了 16384 个处理单元，是目前已知的创纪录数字。对于 32 位浮点操作，它每秒能完成 4.3 亿次加法或 2.16~2.73 亿次乘法 (见下节介绍)。

4. 很好的模块性

多处理机系统中，特别是在采用分布式结构的情况下，每一台处理机都做成一个模块，大量重复设置，便带来一系列与模块性相联系的优点。这种系统不但便于用大规模/超大规模集成电路实现，而且具有极好的结构灵活性 (如可扩充性、可重构性等)。这些特性在改善系统性能上将提供有利的条件，例如，适应不同应用程序和算法保持系统的平衡，对额外任务和高峰负载具备处理潜力，便于在系统上实现多道程序等等。

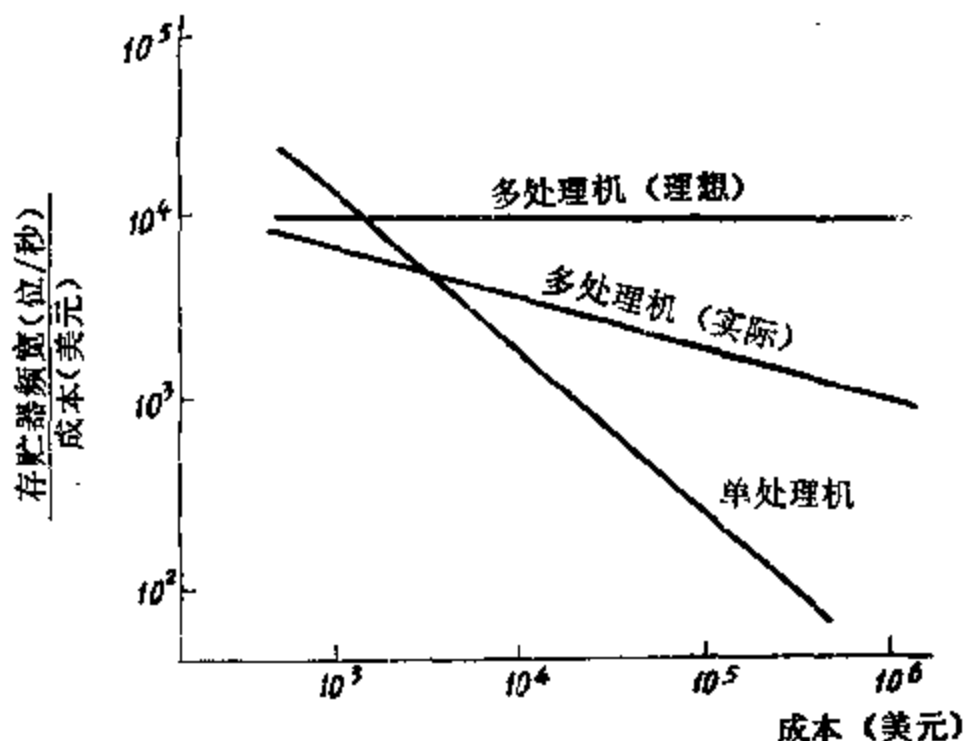


图 7.2 性能价格比与计算机规模的关系曲线

1.3 多机系统的分类

1.3-1 按并行性等级分类

由以上的讨论可知：多机系统是计算机系统结构中发展并行性的产物，因此，用并行性

等级对它们进行分类是很自然的。

(1) 数组并行

前述 n 位并行运算可认为是数组并行, 这是并行性的最低且最基本的等级。通常在这一基础上还可区分计算机系统的四种处理方式:

位串字串——同时只对一个字的一位进行处理, 这是指传统的串行单处理机。

位并字串——同时对一个字的全部位进行处理, 这是指传统的并行单处理机。

位片串字并——同时对许多字的同一位(称位片)进行处理, 这开始进入了并行处理的领域, 例如位串式相联处理机 STARAN^[4, 10]和位平面式阵列处理机 MPP 和 DAP^[17]等。

全并行——同时对许多字的全部或部分位组进行处理, 这是多处理机最普遍的并行方式, 例如全并行相联处理机 PEPE^[4, 10], 全并行阵列处理机 ILLIAC IV^[16], 流水线处理机 TIASC^[11], 多处理机 C.mmp^[9]等。

(2) 存储器操作并行——相联处理机

这里, 存储器操作不但指读/写操作, 而且主要指存储器内部的信息处理。按内容寻址的相联存储器^[10]就正是这样一种带信息处理功能的存储器, 能按位片串字并或全并行的方式对所有存储单元的内容进行一系列操作, 例如, 与给定的、被称为关键字的信息的全部位或部分位特征进行比较、符合、分解(分离多个字的同时符合)等。以相联存储器为核心, 加上必要的中央处理器、指令存储器、控制器和 I/O 接口, 就构成一台以存储器操作并行为特征的相联处理机, 它能以很高的效率对存储器信息进行检索、更新和变换, 还能求解一系列大型计算问题。著名的例子如美国的 STARAN 和 PEPE 系统, 前者包含 32 套模块, 每一套是位片串行、256 字并行的子系统, 后者包含 288 套模块, 每一套都有字长 32 位的并行运算控制部件。

(3) 处理器操作步骤并行——流水线处理机

这里, 处理器操作步骤既可指一条指令的执行步骤, 如取指令、形成地址、取操作数、执行操作等, 又可指操作的执行步骤, 如浮点加的对阶、尾数加、规格化等。把这些步骤分成专门的功能段, 按时间重迭原理组成流水线, 就能实现处理器步骤在多数数据组之间的并行。以这种指令流水线部件和操作流水线部件组成的中央处理器为主, 配上频带与它匹配的存储器、缓冲寄存器和控制器、I/O 接口等, 就构成一台以处理器操作步骤并行为特征的流水线处理机。这种处理机是目前以解决大型数值计算(向量处理)问题为中心任务的巨型计算机的主要型式, 美国的 CDC STAR-100、TIASC 和居于当前最高水平的 CRAY-I^[11]等都是它们的代表。

(4) 处理器操作并行——并行处理机

同样的向量处理, 如数组或矩阵运算, 也可以利用大量重复设置的处理单元并行地完成, 所有处理单元在同一控制器指挥下, 按照同一条指令的要求, 对多数数据组同时进行操作, 这就是处理器操作并行。实现这一级并行性的典型系统是并行处理机。目前普遍采用的一种结构型式以美国的 ILLIAC IV 系统为代表, 64 个处理单元各带 2048 字的内存, 排列成 8×8 的正方形阵列, 每一单元只存在与其四个近邻的有限连接, 这种独特的结构形式是这类处理机又称为阵列处理机(Array Processor)的由来。美国的 MPP, 英国的 DAP 和西德的 EGPA 都是这类处理机的最新代表。在阵列结构以外, 还有以 BSP 科学处理机为代

表的并行处理机，它采用了 16 个处理单元与 17 个存储器模块，共同组成一条数据流水线。

(5) 指令、任务并行——多处理机

并行性不断升级，达到指令、任务、作业等更高等级，它虽然也包含操作并行等较低等级，但原则上与操作级并行是不同的。指令级并行是多个处理机同时对多条指令及其有关的多数据组进行处理，而操作级并行是对同一条指令及其有关的多数据组进行处理，所以前者构成多指令流多数据流计算机，后者构成单指令流多数据流计算机。我们根据 P·H·恩斯洛在文献[2]、[9]中采用的称呼，把前者构成的系统称为多处理机，而把后者构成的系统称为并行处理机。这两种系统作为多机系统的重点将在本章以下二节中分别讨论。

1.3-2 与其它分类法比较

目前普遍采用的一种分类法是 M·J·弗林 1966 年提出的按指令流和数据流对计算机进行分类的方案^[14]。他首先引入了下列定义：

指令流——机器执行的指令序列。

数据流——由指令流调用的数据序列（包括输入数据和中间结果）。

多倍性——在系统最受限制的元件上处于同一执行阶段的指令或数据的最大可能个数。

于是，按指令流和数据流分别具有的多倍性，可将计算机系统分为下列四类：

单指令流单数据流 (SISD)

单指令流多数据流 (SIMD)

多指令流单数据流 (MISD)

多指令流多数据流 (MIMD)

单指令流单数据流系统就是传统的顺序处理计算机。尽管它可能设置多个并行的存储体和多个操作部件，但是只要它的指令部件（系统瓶颈）一次只对一条指令进行译码和只对一个操作部件分配数据，那么它就仍属 SISD 系统。

单指令流多数据流系统以下一节将要介绍的并行处理机为代表，它包含多个重复的处理单元，由单一指令部件按照同一指令流的要求同时向它们分配各自需要的不同数据。如果我们把上述“多倍性”定义中“处于同一执行阶段”理解为一条指令的操作全过程，则流水线处理机也可视为 SIMD 系统一类，其中单一的流水线运算部件把操作全过程分为许多操作步骤，同时对多个数据进行加工。以资源重复为基础的相联处理机也是属于这一类。

多指令流单数据流系统在实际上代表何种计算机，存在着不同的看法。有的文献上把流水线处理机分到这一类。如果把指令流水线看作多指令部件，也许能提供一种解释；但它又是和操作流水线的多数据流工作方式分不开的。

多指令流多数据流系统则是实现作业、任务、指令、数组各个级别全面并行的计算机系统，将在本章第 3 节中讨论。

把 § 1.3-1 节中按并行性等级进行的分类与本小节中介绍的按指令流和数据流的弗林分类法进行比较，可以看出，二者在实质上是一致的。这是因为前者也是把着眼点放在能否实现指令级并行这一关键问题上。它划分的五种并行性等级中，第一种是反映数据流的情况，后四种是反映指令流的情况。而后四种中，前三种的并行性都低于指令级，属于 SIMD 系

统，最后一种并行性达到指令级，才属于 MIMD 系统。按并行性对计算机系统进行分类，还可附带得到三点有用的结果：第一，它对 SIMD 系统包含的过多类型作了较细的划分，反映了它们不同的工作原理。第二，它反映 MIMD 系统又是可以进一步划分的，例如指令级并行、任务级并行（多任务）和作业级并行（多道程序）等，它们实际上要求系统具有不同的特性。第三，它说明 MISD 系统如果不是不存在，也是实际中很少见的，这是因为这种系统要求指令级并行，却不在数组级全并行，与通常高级并行性总是包含低一级并行性的做法不甚符合。

此外，还存在多种其它的分类方案，也都反映了不同的并行性等级、区分了资源重复和时间重迭（流水线）等不同的情况，本书就不一一介绍了。关于本节的一些较详细的内容，读者可参阅文献[7]。

§2 并行处理机

本节首先讨论并行处理机的特点和工作原理；进而以 ILLIAC IV 为例介绍阵列处理机的主要结构，举例说明这种结构对算法的适应性，还指出了这种结构型的最新发展情况。作为并行处理机的关键组成部分，重点讨论 SIMD 互连网络的结构原理。

2.1 并行处理机的特点与组成

2.1-1 并行处理机的工作原理和组成

从前面关于并行性等级的讨论可知：并行处理机是依靠在操作一级实现并行处理来提高系统速度的，它是属于单指令流多数据流计算机一类。并行处理机的工作原理可从图 7.3 和

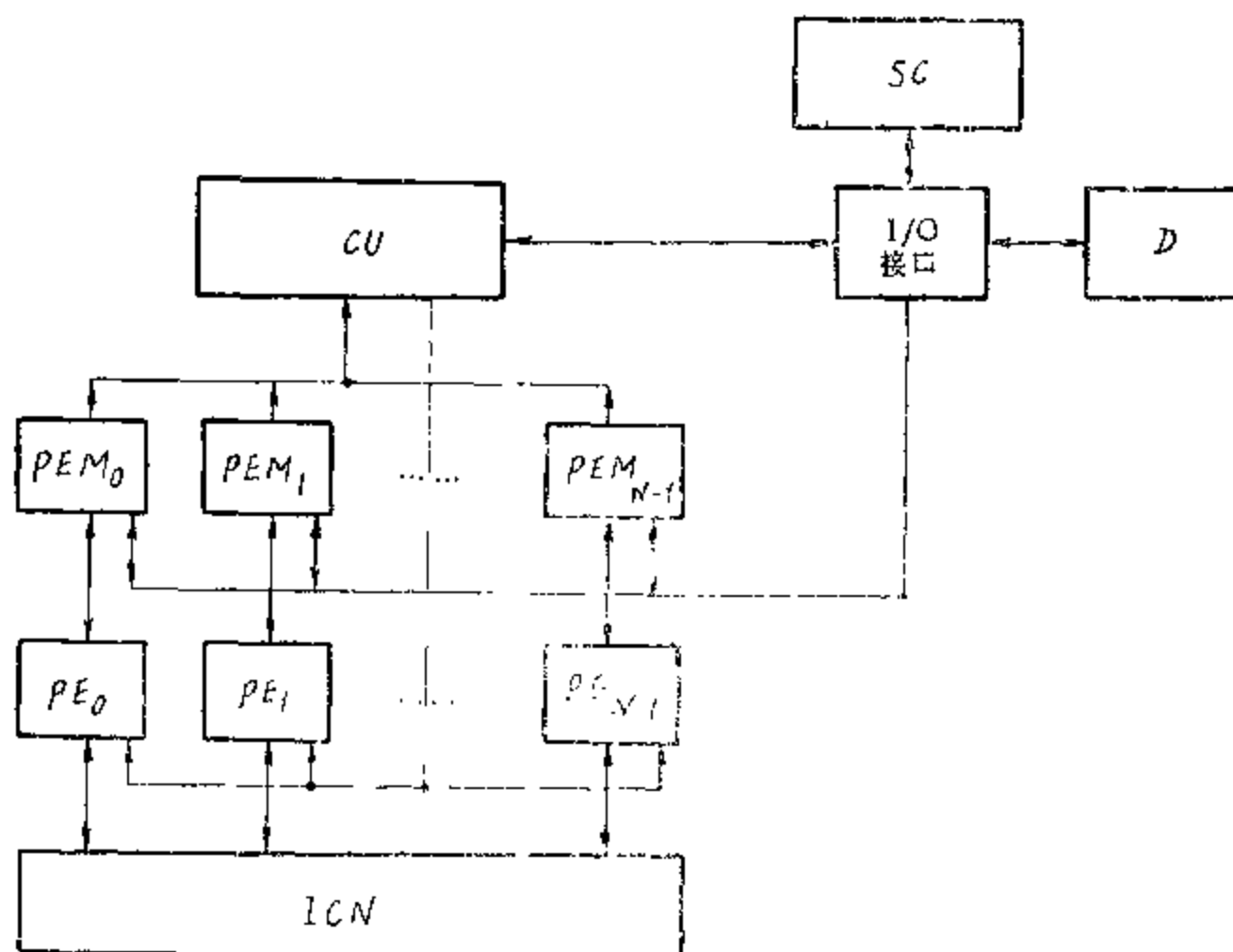


图 7.3 具有分布存储器的并行处理机结构框图

图 7.4 的结构框图得到说明。它们的共同特点是重复设置许多个同样的处理单元 PE，按照一定的方式相互连接，在统一的控制部件 CU 作用下，各自对分配来的数据并行地完成同一条指令所规定的操作。从控制部件来看，指令的执行顺序还是和通常的高性能单处理机一样，基本上是串行处理加先行控制；差别只在于它把指令分成两类：一类是控制指令和只适合于串行处理的操作指令，留给自己来完成；但如果遇到的是另一类适合于并行处理的指令，就“播送”（表示一个源送往多个目的地）给所有的处理单元。但是，只有那些处于“活动”状态的处理单元才并行地对各自的数据完成这一操作。至于存储器，则反映了图 7.3 和图 7.4 的主要差别。图 7.3 表示具有分布存储器的并行处理机结构，它把存储器分成二部分，一部分集中在专施管理功能的主机 SC 内，作常驻操作系统之用；另一部分 PEM 则分布地设置在各个处理单元一起，用来存放程序和数据。为了有效地进行高速处理，数据应在各处理单元之间合理分配，使它们都可以主要依靠自身存储器中的数据进行运算。但是，以后将可看到，无论如何，处理单元之间的数据交换仍是必要的。因此在处理单元（或存储器）之间必须设置一个互连网络 ICN，作为数据交换的通路。图 7.4 中具有集中存储器结构的并行处理机的做法则有所不同。除了主机内的那一部分存储器（图中未画出）以外，各个处理单元的本地存储器也集中放在一起，构成统一的主存 MM，经过互连网络 ICN 被全部处理单元 PE 所共享。输入/输出设备 I/O 和外存储器 SM 也可以经过 I/O 通道 (I/O-CH) 与这个共享存储器交换信息。

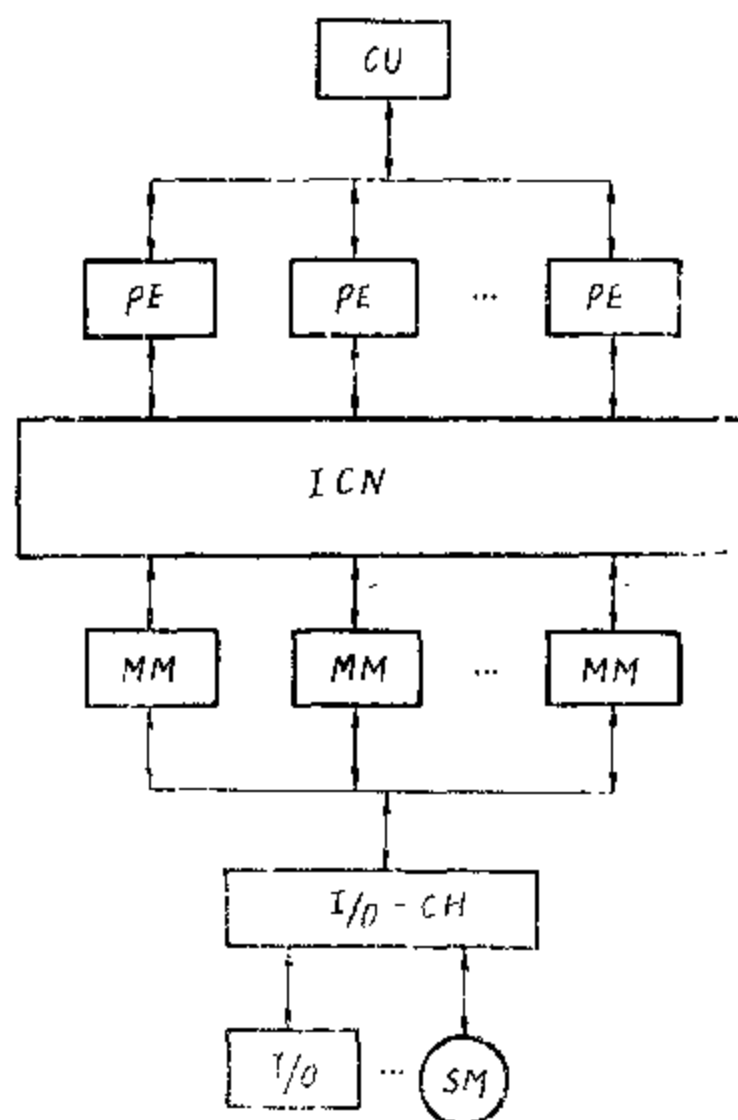


图 7.4 具有共享存储器的并行处理机结构框图

2.1-2 并行处理机的专用性特点

上述并行处理机的单指令流多数据流处理方式和由它产生的特殊系统结构是以诸如有限差分、矩阵、信号处理、线性规划等一系列计算问题为背景而发展起来的。这些问题的共同特点是循各种途径归结为数组和向量处理。并行处理机的多操作单元对向量所包含的各个分量同时进行运算，正是它获得很高处理速度的主要原因。与同样擅长于向量处理的流水线处理机相比，并行处理机依靠重复的处理单元，而不是依靠时间重迭来实现并行运算，而且它的每一个处理单元都要同等地负担多种运算功能，相当于多功能流水线部件（如 TIASC），效率自然比多个单功能流水线部件（如 CRAY-I）要低一些。所以只有在硬件价格大幅度下降，加上系统结构的改进，并行处理机才能具有较好的性能价格比。但是，并行处理机在机间互连方式的灵活性上比固定结构的单功能流水线部件要好，在相当一部分专门问题上的工作效率要比流水线处理机高得多。所以如果习惯上把流水线处理机归于通用计算机一类，那么并行处理机实际上可被视为专用计算机，它以一定的专用算法为背景。这是其特点之一。

并行处理机的机间连接,与按多指令流多数据流方式工作的多处理机相比,具有较固定的结构,它直接与一定的算法相联系,其效率取决于在多大程度上把计算问题归结为向量数组处理。当然,我们应该在这个前提下通过改进系统结构和制定并行算法,使并行处理机可能适应的计算问题类型尽量广一些、多一些,也就是说,应该把系统结构研究和算法研究结合起来。这是并行处理机的第二个特点。

并行处理机的第三个特点是:它的效率只有在向量数组处理上才能充分发挥出来,所以它除了要有一个功能很强的控制部件以外,还必须和一个高性能单处理机配合工作。控制部件实际上也是一台标量处理机,它和专门担负向量处理功能的处理单元阵列并行工作,以很高的传送频率和后者交换指令和数据。系统的管理功能则由那一台高性能单处理机担负。所以实质上是三者共同组成一个多处理机系统,而处理单元阵列可以说只是它的一个向量处理部件。

2.2 阵列处理机的结构

2.2-1 ILLIAC IV 的结构原理

阵列处理机是并行处理机最普遍的一种结构型式,它的基础是六十年代初美国西屋电器公司在 SOLOMON 机设计中提出来的网状结构,后来在六十年代末依里诺大学设计 ILLIAC IV 处理机中得到了改进和采用,称为阵列结构。图 7.3 也可以说就是阵列处理机的原理框图,因为它基本上是参照 ILLIAC IV 的结构画出来的,其中用作管理计算机 SC 的是一台 Burroughs 公司的 B-6700; D 是一台 10^6 位的大容量并行读写磁盘存贮器;二者都通过 I/O 接口与处理单元阵列相联系。ILLIAC IV 的最初设计规模是 4 个相互联系的处理单元阵列,每一阵列包含 64 个处理单元和一个控制部件,但是到后来只实现了一个阵列,而这个阵列的规模已经达到了惊人的程度(每一个处理单元包含十万个以上的分立元件,装成 12000 个开关电路) [18]。

ILLIAC IV 的处理单元阵列较详细地表示在图 7.5 中。64 个单元 $PE_0 \sim PE_{63}$ 排列成 8×8 的方阵,每一个 PE_i 只和其东、南、西、北四个近邻 $PE_{i+1}(\text{mod } 64)$ 、 $PE_{i+8}(\text{mod } 64)$ 、 $PE_{i-1}(\text{mod } 64)$ 和 $PE_{i-8}(\text{mod } 64)$ 有直接连接。步距不等于 ± 1 或 ± 8 的机间通信可以用软件方法寻找最短路径进行,例如,向右进 5 步可以用右进 8 步加左进 3 步实现。可以看出,任意两个处理单元之间的最短距离不会超过 7 步。图 7.5 把全部处理单元画在一条直线的方向上是为了表明 $PE_0 \sim PE_{63}$ 首尾相连构成一个闭合环形的情形;同时,也表示出有 64 位公共数据总线和近 260 根控制线把 64 个 PE 及 PEM 和控制部件连接起来;此外,还有 1024 位数据线将 PEM 与 I/O 开关相连接。如果把

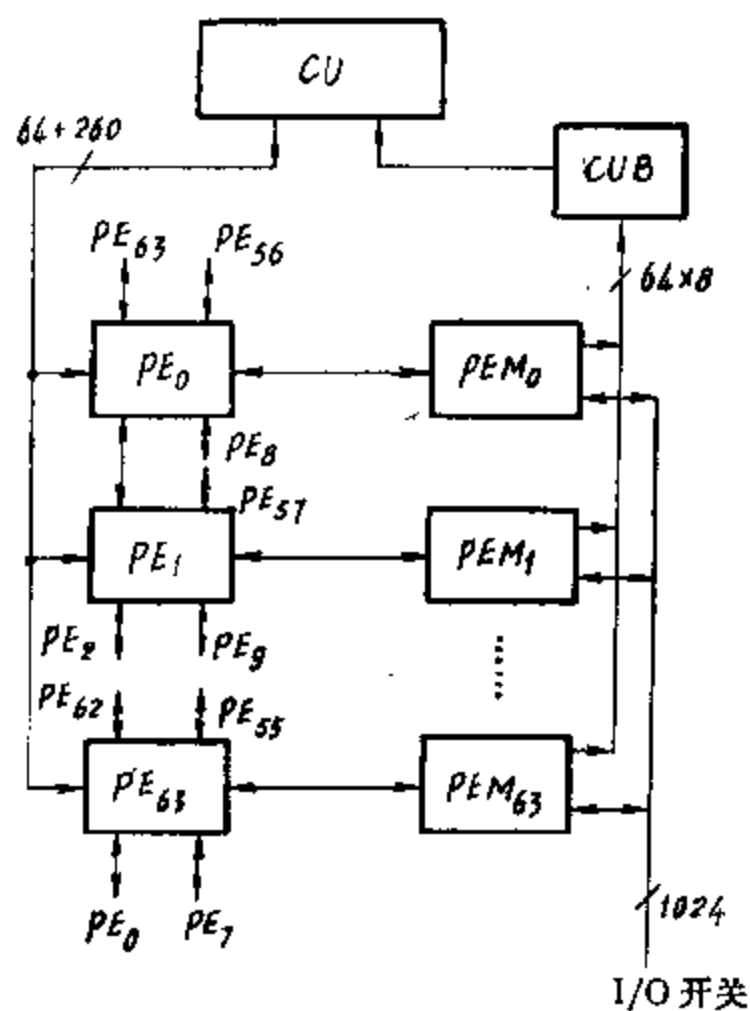


图 7.5 ILLIAC IV 处理单元阵列原理框图

64 个 PE 画成一个方阵,则如图 7.6 所示,呈闭合螺旋线形状,称闭合螺旋线阵列。图中,PU 代表

处理部件，是处理单元 PE 和其附属存贮器 PEM 以及存贮器逻辑部件 MLU 的总称。

2.2-2 处理单元

处理单元 PE 是向量数组处理的运算部分，它可对 64 位、32 位或 8 位操作数进行多种操作，包括定点运算以及 64 位或 32 位浮点运算。这等于将 64 个处理单元的硬件当作 64 个 (64 位)、128 个 (32 位) 或 512 个 (8 位) 处理单元发挥作用。并行的加法速度是每秒 10^{10} 次 8 位定点加法或 150×10^6 次 64 位浮点加法。操作数的来源有四：从 PE 本身的寄存器，从阵列存贮器，从 CU 的数据总线或是从 PE 的四个近邻。

处理单元的结构框图如图 7.7 所示。它包含下列部件：

(1) 四个用来存放操作数和结果的 64 位寄存器 RGA、RGB、RGR 和 RGS。RGA 是累加寄存器，存放第一操作数和结果；RGB 是操作数寄存器，存放第二操作数；RGR 是被乘数和互连寄存器，处理单元之间的数据直接交往都是通过它来完成的；RGS 是通用寄存器，可给程序员用来暂存中间结果。

(2) 三种运算部件，包括一个加法/乘法器 AU、一个逻辑单元 LU 以及一个用全移位器实现的移位单元 SU。

(3) 二种变址部件，包括一个 16 位变址寄存器 RGX 和地址加法器 ADA。有效地址通过存贮器地址寄存器 MAR 送往 MLU。

(4) 一个 8 位的模式寄存器

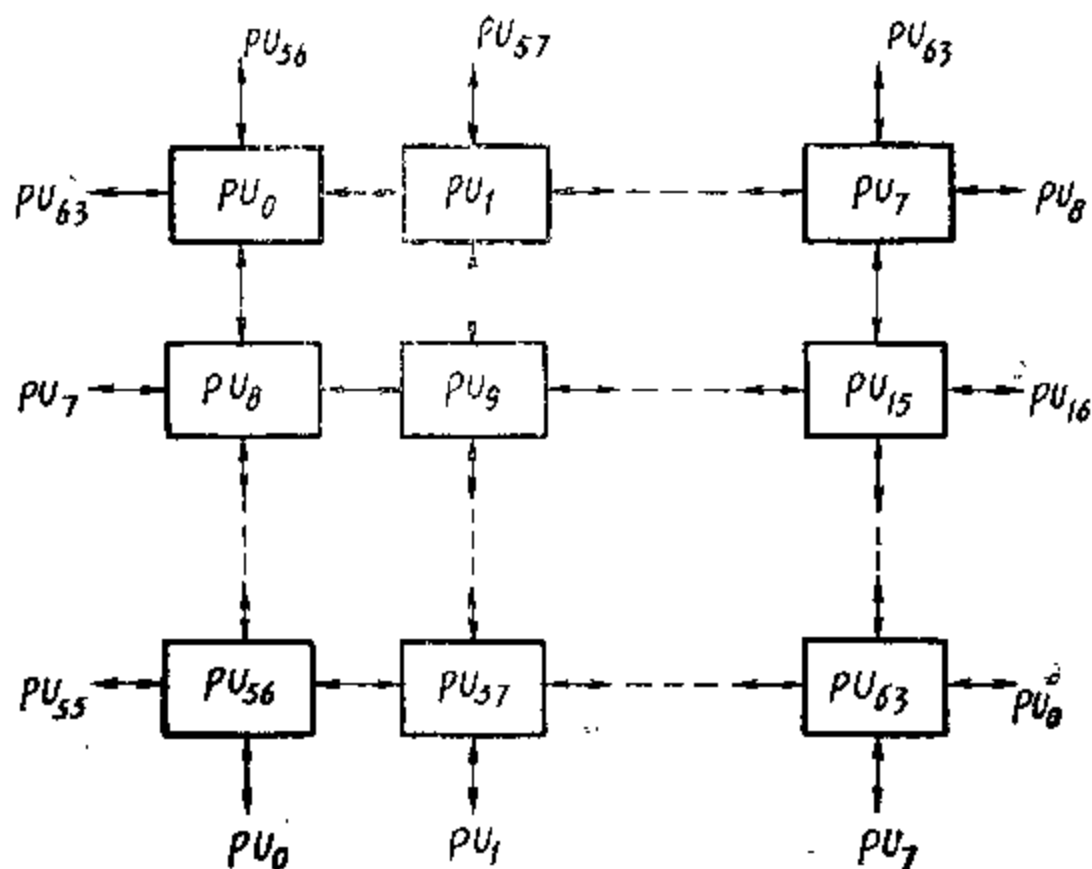


图 7.6 ILLIAC IV 处理部件的相互连接

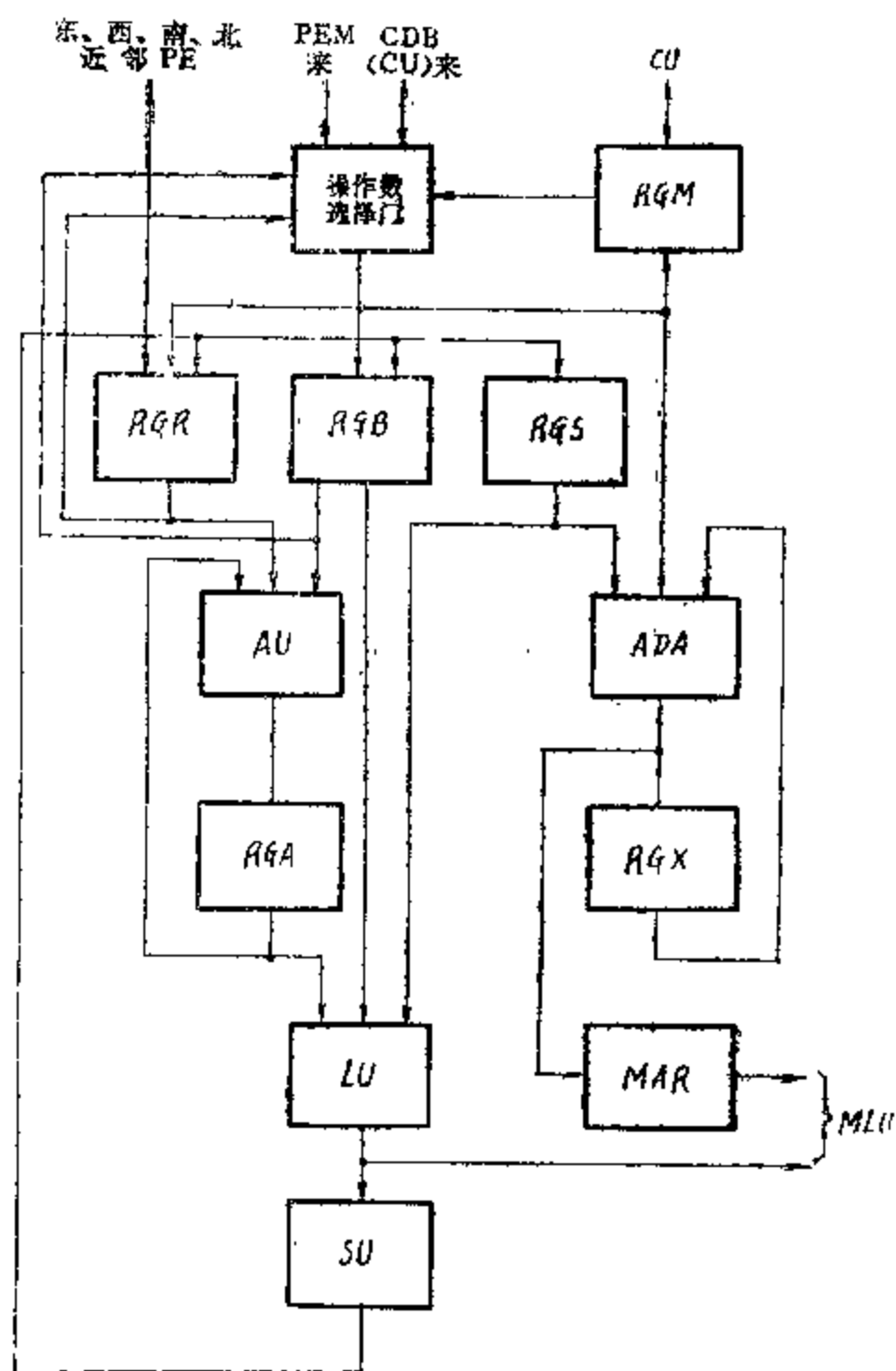


图 7.7 ILLIAC IV 处理单元原理框图

RGM, 其中 E 和 E1 是“活动”标志位, 用来控制 RGA、RGS 和 MLU 中存贮信息寄存器 MIR 的工作, 使只有处于活动状态的处理单元才执行指令的共同操作, 此外, E 还控制 RGX。当 PE 按 32 位模式运算时, E 和 E1 是互相独立的。F 和 F1 保存运算结果出错 (上溢、下溢) 标志, G、H、I、J 位保存测试结果。RGM 经常处于 CU 的监督之下, 一旦出错, 就发出 CU 陷阱中断。模式寄存器对增强阵列处理机的功能和结构灵活性发挥着很大的作用。

2.2-3 控制部件

控制部件的功能有五:

- (1) 对指令流进行控制和译码, 包括执行一整套标量操作指令;
- (2) 向各处理单元发出执行数组操作指令所需的控制信号;
- (3) 产生和播送所有处理单元公共的地址部分;
- (4) 产生和播送所有处理单元公共的数据;
- (5) 接收和处理各处理单元 (计算出错时)、系统 I/O 操作以及 B-6700 产生的陷阱信号。

其原理框图如图 7.8 所示。它和处理单元存贮器 PEM 的联系是通过两个各 64 字的快速缓冲器建立的, 一个是局部数据缓冲器 LDB, 另一个是指令缓冲器 IB, 后者由相联存贮器 CAM 控制, 具有先行功能。指令缓冲器能保存多至 128 条指令, 足够供大多数程序循环使用。CU 向 PEM 取指令, 每次以 8 个字 (16 条指令) 为单位。缓冲器加载的策略如下: 当指令计数器正处在一个指令块 (8 个字) 的半途时, 就要开始执行取下一块的动作, 而不考虑下面会不会遇到转移指令。如果发现下一块已经在缓冲器内, 则无需采取行动; 如若不然, 就要向阵列存贮器发出读命令。新读出的指令块就把最早执行过的那一块指令当作最少需要者而排挤掉, 这符合“最早历史”原则。从阵列存贮器读出指令块, 并送到 CU, 考虑到传输延迟, 需要占用 3 个存贮周期的时间。由于正在执行中的指令块还剩余 8 条指令, 故取新指令块的时间可以和剩余指令的执行时间重迭起来。

所有指令都是 32 位长。指令系统与传统的计算机相似, 但有两点例外: 一是互连指令, 负责与相邻处理单元交换信息; 二是有处理单元使自己的模式寄存器置为活动或不活动的指令。指令分为二类: 由 CU 本身执行的 CU 类指令 (如变址、转移等), 以及由 CU 译码、然后控制 PE 执行的 PE 类指令。指令逐条地从指令缓冲器取至指令前进站 ADVAST。

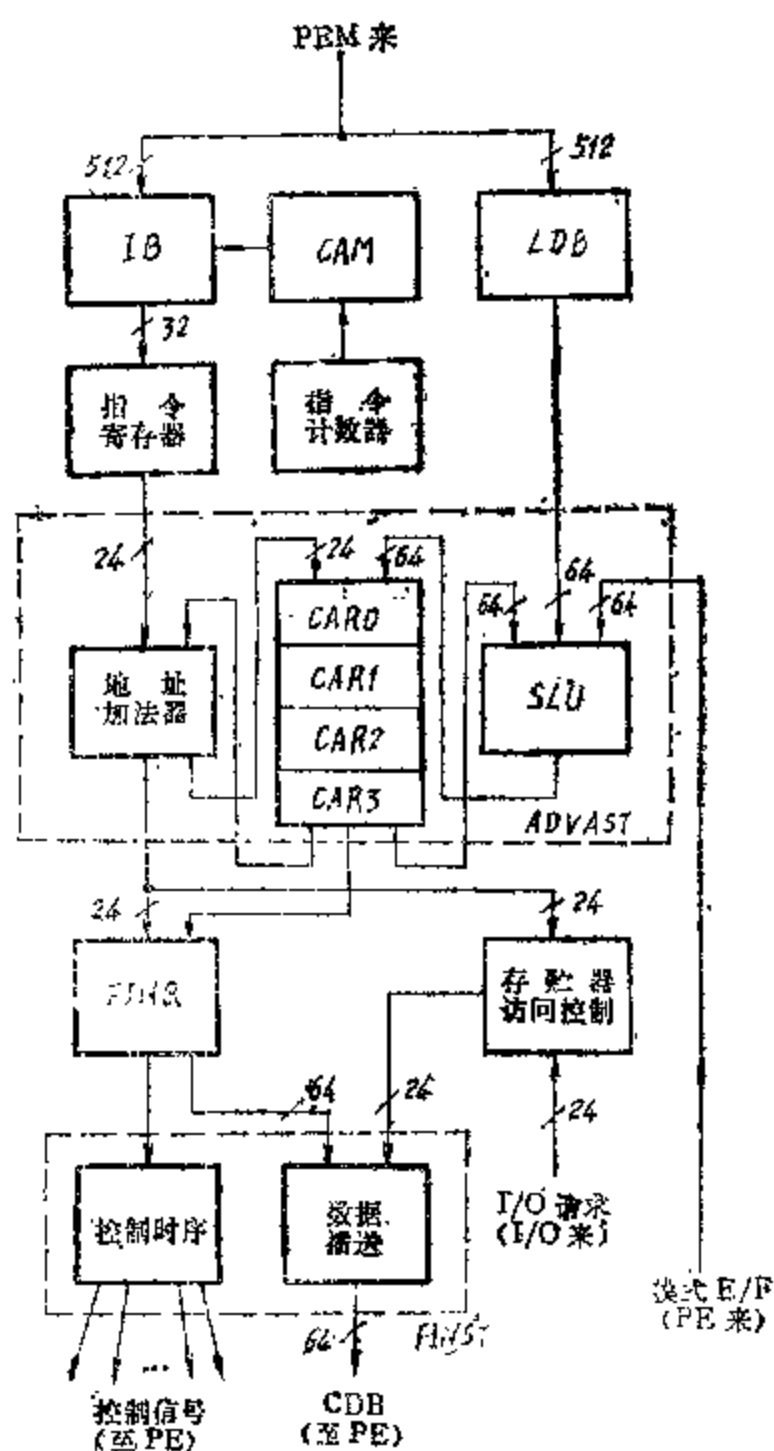


图 7.8 ILLIAC IV 控制部件原理框图

该前进站是 CU 的处理部分,它包含 4 个累加寄存器 $CAR_0 \sim 3$,一个 24 位地址加法器和一个 64 位移位逻辑部件 SLU。CAR 用来寄存变址信息、局部运算数据和播送的信息。SLU 完成定点加、减和布尔操作,其它复杂操作则交由各处理单元去完成。进入 ADVAST 的如果是 CU 类指令,就在其中译码和执行;如果是 PE 类指令,就形成地址和操作数,送至最后排队器 FINQ,等待向各处理单元传送。传送时经过指令最后站 FINST,它用来控制地址或数据的播送以及在执行期间保存 PE 指令。最后排队器的设置是为了使 CU 指令和 PE 指令能重迭执行。

局部数据缓冲器和指令缓冲器不同,它既可以读取 8 个字的数据块,也可以读取单独的字。此外,还可以从每个处理单元的模式寄存器各读一位,拼成一个 64 位的字,直接读至 ADVAST 的移位逻辑单元。这样 CU 便可感知所有处理单元的状态,作为根据以便对它们进行个别控制。

2.2-4 阵列存贮器

每一个处理单元 PE 都有自己的 2048×64 位的处理单元存贮器 PEM,这 64 个 PEM 联合组成阵列存贮器,存放数据和指令。整个阵列存贮器可以接受 CU 的访问,读出 8 个字的信息到控制部件缓冲器 CUB 中,还可经过 1024 位的总线与 I/O 开关相连(见图 7.5),但是每一个处理单元只能访问自己的存贮器。如果把 64 个 PEM 看成列,每一个 PEM 本身看成行,那么整个阵列存贮器就如同一个二维访问存贮器,CU 对它是按列访问,PE 对它是按行访问。访问是经过存贮器逻辑单元 MLU 进行的,它包含存贮信息寄存器 MIR 和必要的逻辑部分,作为 PE 向 PEM 的接口,实现 PEM 和 CU 及 PEM 和 I/O 之间的信息传送。

阵列存贮器的这种安排带来很大好处,它使分布在处理单元存贮器中的指令或数据可先送往控制部件 CU,再经公共数据总线 CDB 由 CU 播送到全部 64 个处理单元中。这不但节省了存贮空间,而且允许公共数据存取与其它操作在时间上重迭。

存贮器的另一独特之点是它的双重变址机构。除了控制部件的公共变址以外,每一个处理单元还可单独变址。最终操作数地址对 i 处理单元来说可决定如下:

$$a_i = a + (b) + (c_i)$$

式中 a 是指令指定的基地址;

(b)是 CU 中央变址寄存器内容;

(c_i)是 i 处理单元局部变址寄存器内容。

这种安排增加了各处理单元存贮器数据分配的灵活性,对于分别处理矩阵的行和列以及其它多维数据结构是很有效的。

阵列存贮器的加载是经过 I/O 开关进行的,它是阵列存贮器、并行读写磁盘存贮器和主机 B-6700 三者之间的接口(见图 7.3)。阵列存贮器可接收作为阵列外存的并行磁盘的信息,也可接收 I/O 缓冲存贮器的信息;而经 B-6700 汇编后的程序先送到 I/O 缓冲存贮器,再由它既可送入并行磁盘,也可送入阵列存贮器中。

2.3 阵列处理机的算法举例

为了具体说明 ILLIAC IV 阵列结构的特点和一些附加功能的作用,下面举几个算法的

例子 15.01。

2.3-1 有限差分问题

ILLIAC IV 的基本结构是一个包含环形移位连接的二维阵列，这种结构特别适应于计算在网格上定义的有限差分函数。在每一网格点上的函数值受其邻近点的直接影响，可以通过求平均值的方法多次迭代，逐次逼近其最终的平衡值。假定一平面网格在 x 和 y 方向上的间隔均为 h ，则任一网格点 (x, y) 上的函数值可按下式由其四周邻近点的函数值计算之：

$$U(x, y) = \frac{U(x+h, y) + U(x, y+h) + U(x-h, y) + U(x, y-h)}{4} \quad (7.2-1)$$

在解决物理场的问题时，如果将描述平面场的拉普拉斯方程

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0 \quad (7.2-2)$$

中的二阶偏导数表示为差分形式，

$$\left. \begin{aligned} \frac{\partial^2 U}{\partial x^2} &= \frac{U(x+h, y) - 2U(x, y) + U(x-h, y)}{h^2} \\ \frac{\partial^2 U}{\partial y^2} &= \frac{U(x, y+h) - 2U(x, y) + U(x, y-h)}{h^2} \end{aligned} \right\} \quad (7.2-3)$$

并代入原方程，即可得到上述计算公式。实际计算时，利用张弛法进行。先假定网格边缘点的函数值作为初始条件是已知的，而内部各点的函数值都等于零。然后根据上述有限差分公式多次迭代求 $U(x, y)$ 值，直至连续二次迭代所求值相差很小为止（已知迭代过程是收敛的）。ILLIAC IV 的处理单元正好构成这样的网格，把内部网格点分配给各个处理单元，则上述公式的计算可以并行地完成，几十倍地提高处理速度。实际问题中遇到的内部网格点数目往往是很大的，就要分成许多子网格，才能在 ILLIAC IV 上求解。

2.3-2 矩阵问题

矩阵加是最简单的情形。假定二个 8×8 的矩阵 A 、 B 相加，只需把 A 和 B 居于相似位置的一对分量存放在同一 PEM 内，且令 A 的分量在全部 64 个 PEM 中存放的单元地址码均为 α ， B 的分量单元地址码均为 $\alpha+1$ （如图 7.9 所示），则只需下列三条 ILLIAC IV 的汇编指令就可一次实现矩阵相加：

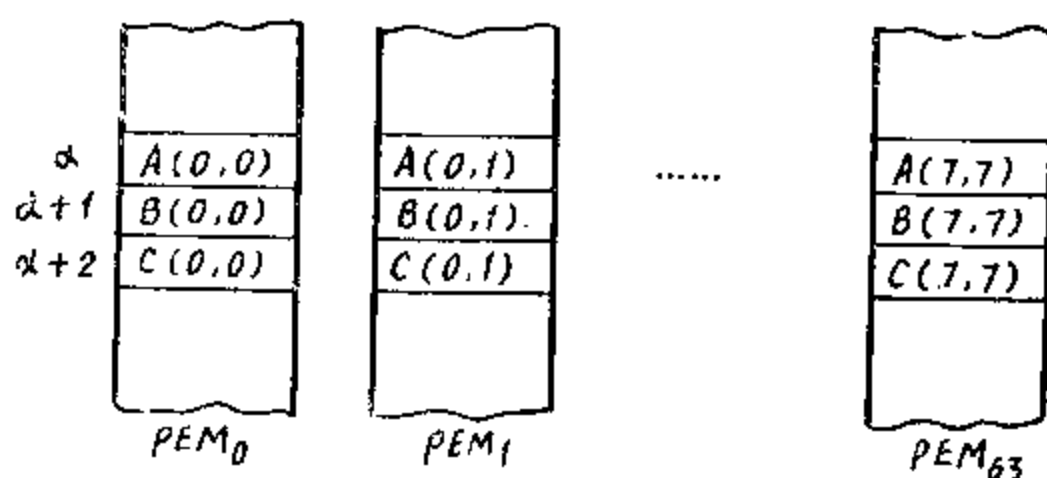


图 7.9 矩阵相加存储器分配举例

LDA ALPHA ; 全部 (α) 由 PEM 送 PE 的累加器 RGA
ADRN ALPHA+1 ; 全部 $(\alpha+1)$ 与 (RGA) 进行浮点规舍加，结果送 RGA
STA ALPHA+2 ; 全部 (RGA) 由 PE 送 PEM 的 $\alpha+2$ 单元

其次是矩阵乘法。设 A, B 和 C 为三个 8×8 的二维矩阵。若给定 A, B, 则为计算 $C = A * B$ 的 64 个分量 $C(I, J)$, $0 \leq I \leq 7$, $0 \leq J \leq 7$, 可利用下列公式:

$$C_{ij} = \sum_{k=0}^7 a_{ik} b_{kj}, \quad 0 \leq i, j \leq 7$$

(7.2-4)

如果在 SIMD 计算机上求解这个问题, 可执行用 FORTRAN 语言书写的下列程序:

```

      DO 10 I=0, 7
      C(I, J)=0.
      DO 20 K=0, 7
20    C(I, J)=C(I, J)+A(I,
      K)*B(K, J)
10  CONTINUE
  
```

类似的程序在 SISD 计算机上要用 K, I, J 三重循环才能完成, 每重循环执行 8 次, 共需用 512 次加、乘时间 (不考虑其它控制类指令执行时间)。而在 SIMD 计算机上执行上面的程序, 如果利用 8 个处理部件并行计算, 则 J 循环只需一次即可完成, I, K 循环照旧, 因此, 可提高速度至 8 倍, 即只需 64 次加、乘时间。程序流程图如图 7.10 所示,

表面上看, 这个程序在每个处理部件内部的执行过程和传统的 SISD 计算机是一样的, 但是由于 8 个处理部件执行同一套指令, 并行操作, 实际解决问题的方式是不同的: 第一, 控制部件执行的 PE 类指令表面上是标量指令, 但实际上等效于向量指令, 如向量取、向量存、向量加、向量乘等。第二, 执行这个程序要求 A, B, C 向量的各分量在处理单元存储器中的分布方案如图 7.11 所示。作乘法时, 操作数 $B(K, J)$ 都从本处理部件的 PEM 中读取; 但被乘数 $A(I, K)$ 对所有处理部件 $0 \leq J \leq 7$ 都是同样的, 一般都不在本部件存储器内, 所以要利用阵列处理机的“播送”功能, 把一次 K 循环中的公共系数 $A(I, K)$ 取出后送到控制部件, 再播送到全部 8 个处理单元的 RGA 中去。

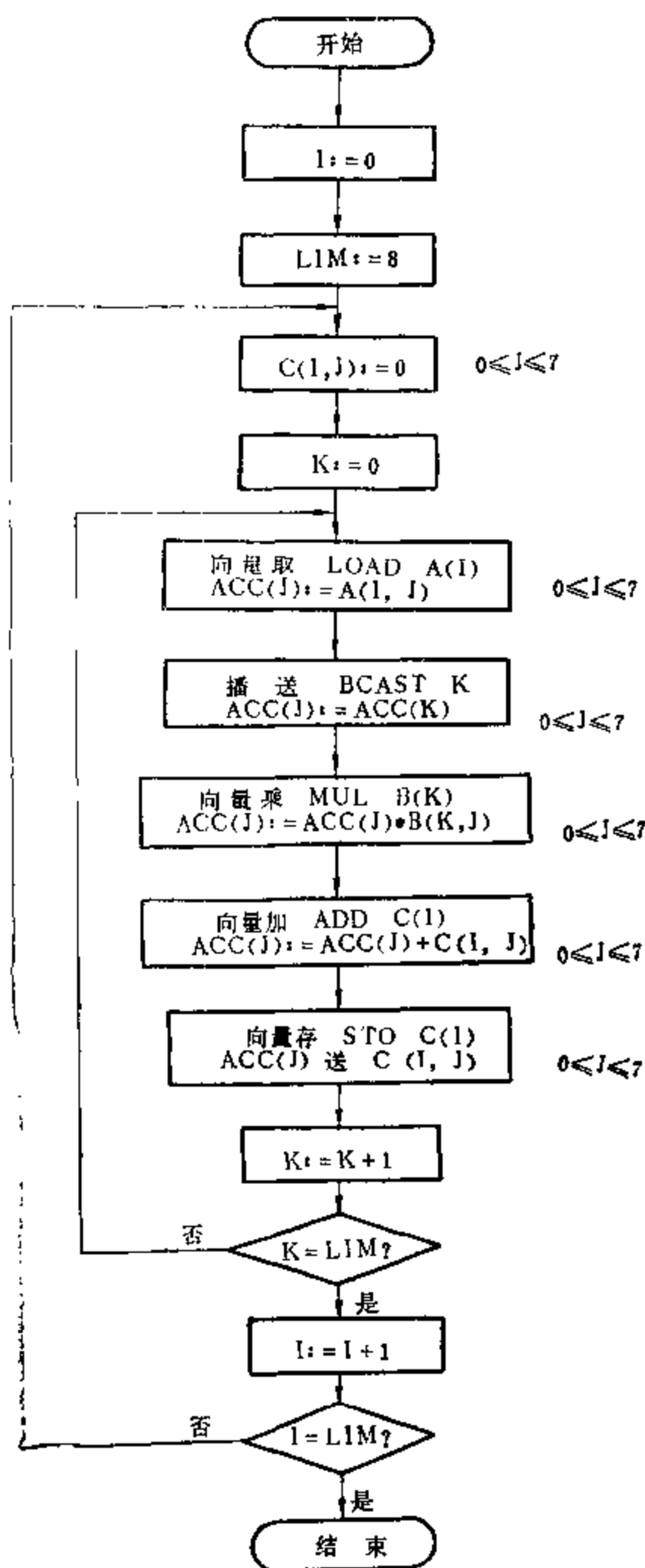


图 7.10 矩阵乘程序流程图

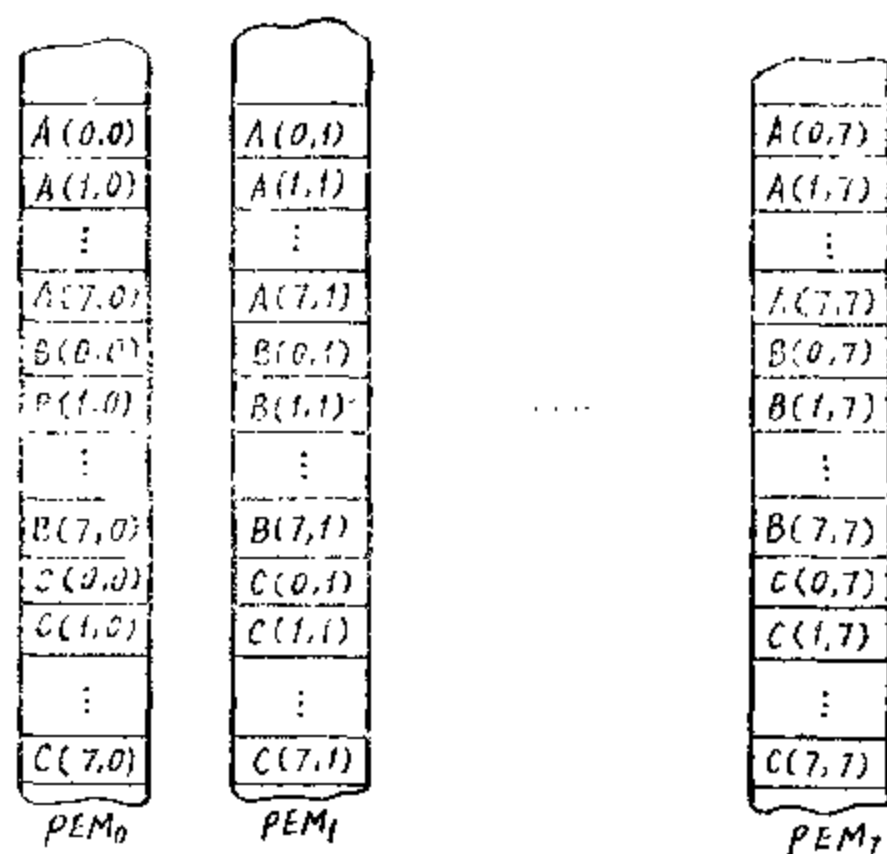


图 7.11 矩阵乘存贮器分配举例

2.3-3 累加和

这是一个将 N 个数的顺序相加过程变为并行相加过程的问题，用得着处理单元中的活动标志位。为叙述方便，取 $N=8$ 。假定有八个数 $A(I)$ ， $0 \leq I \leq 7$ ，顺序累加，可写成下列 FORTRAN 程序：

```

      C(-1)=0
      DO 10 I=0, 7
10    C(I)=C(I-1)+A(I)

```

这是一个串行程序，在 SISD 计算机上，它要求 8 次加法时间。如果在并行处理机上，采用成对递归相加的算法，则只需 $\log_2 8 = 3$ 次加法时间就够了。首先，原始数据 $A(I)$ ($0 \leq I \leq 7$) 存放在 8 个 PEM 的 α 单元中，然后按照下面的步骤求累加和：

- 第一步 置全部 PE 为活动；
- 第二步 全部 $A(I)$ ($0 \leq I \leq 7$) 从 PEM 的 α 单元读入相应 PE 的 RGA；
- 第三步 令 $K=0$ ；
- 第四步 全部 PE 的 (RGA) 转送 RGR；
- 第五步 全部 PE 的 (RGR) 经过互连网络向右传送 2^k 步距；
- 第六步 $j=2^k-1$ ；
- 第七步 置 PE_0 至 PE_j 为不活动；
- 第八步 活动的 PE 执行 $(RGA) := (RGA) + (RGR)$ 操作；
- 第九步 $K := K + 1$ ；
- 第十步 如 $K < 3$ ，转回第四步，否则继续往下执行；
- 第十一步 置全部 PE 为活动；
- 第十二步 全部 PE 的 (RGA) 存入相应 PEM 的 $\alpha+1$ 单元中。

表 7.1 列出各次循环中 PE 几个有关寄存器内容的变化情况。为简单起见， $A(0) \sim A(7)$ 等在表中就用 0~7 等数字代表；且第五步中 PE 的 (RGR) 右移超出 PE_7 的内容就不必表示

表 7.1 累加和 $\sum_{I=0}^7 A(I)$ 的各步计算结果

循环	K=0				K=1			K=2		
步骤	第二步	第五步	第七步	第八步	第五步	第七步	第八步	第五步	第七步	第八步
寄存器 PE	RGA	RGR	活动位	RGA	RGR	活动位	RGA	RGR	活动位	RGA
0	0		0	0		0	0		0	0
1	1	0	1	1+0		0	1+0		0	1+0
2	2	1	1	2+1	0	1	2+1+0		0	2+1+0
3	3	2	1	3+2	1+0	1	3+2+1+0		0	3+2+1+0
4	4	3	1	4+3	2+1	1	4+3+2+1	0	1	4+3+2+1+0
5	5	4	1	5+4	3+2	1	5+4+3+2	1+0	1	5+4+3+2+1+0
6	6	5	1	6+5	4+3	1	6+5+4+3	2+1+0	1	6+5+4+3+2+1+0
7	7	6	1	7+6	5+4	1	7+6+5+4+3+2+1+0		1	7+6+5+4+3+2+1+0

了，这是因为如右移步距为 $2^k(\text{mod } N)$ ，则它们应移入 PE_0 至 PE_j ，但既然这些 PE 在第七步中将要置为不活动状态，故无论它们的 RGR 接受什么内容，都是不起什么作用的。这个并行累加过程也可形象地用图 7.12 来表示，

框中标明各处理单元 RGA 中的相加结果，划有影线的表明不活动。

这个例子说明：虽然经过变换，原来串行的程序也能在 ILLIAC IV 上执行，但由于屏蔽了一部分处理单元，降低了它们的利用率，所以速度提高的倍数也就不等于处理单元的个数 N ，而只是 $N/\log_2 N$ 。

2.4 并行处理机的近期发展

2.4-1 阵列处理机的评价

由 ILLIAC IV 所开创的并行处理机阵列结构，作为把大量处理单元连接起来的一种结构型式，连

同它的 SIMD 工作原理，被实践证明是有生命力的。下节将可看到，在各种 SIMD 互连网络中，阵列结构是最简单的一种。虽然它只能实现有限的连接，但对于一些典型的应用问题仍是高效的，而换来的好处是比较简单，较易于构成规模很大的系统。

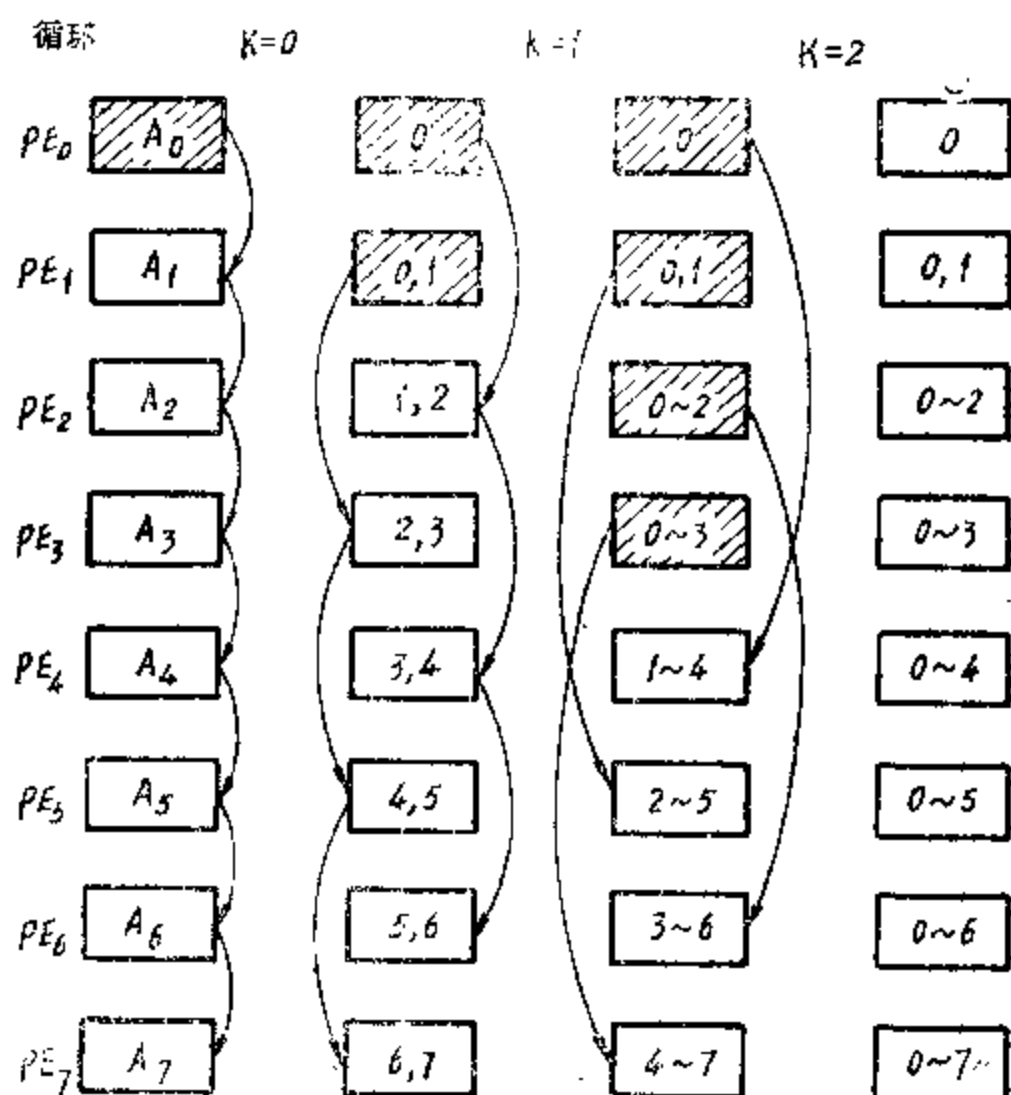


图 7.12 并行处理机上累加和计算过程示意图

当然, ILLIAC IV 本身也存在一些问题;需要在发展阵列处理机的过程中加以考虑和解决。主要有以下几点:

(1) 算法必须专门设计,以适应处理单元阵列的结构特性,其中影响较大的是:相对固定的连接方式;有限而固定数目的算术单元以及 SIMD 操作原理等。上节已经看到,这类算法包括二维有限差分问题和矩阵运算;此外,象快速傅立叶变换、卷积、相关等处理很大规模数组的问题都是阵列处理机能发挥高效率的场合。程序的向量化是设计这些算法的主要目标,以便能够直接利用并行处理机的数组运算能力。

(2) ILLIAC IV 的阵列存贮器各有一个存贮模块分属每一个处理单元,这种分布式存贮器是阵列处理机的典型结构型式。与此相适应,应当对存贮器中的数据分布提出一定的要求,使读取数组元素的不同组合(例如矩阵的行、列、对角线)时不致产生存贮器访问冲突,而又不需要对数据进行复杂的重新排列。这两个要求是相互矛盾的。如果按照数组的自然顺序存放数据,往往不能完全满足第一个要求;如果为了满足第一个要求而把数据错开位置存放,则又不能很好满足第二个要求。特别对于 ILLIAC IV,每一个处理单元只能直接读取自身存贮器中存放的数据;而要在不同处理单元之间交换数据,又只能实现一个处理单元与其四个近邻的直接连接,因此数据的重新排列是很费时间的。但是,这种分布式阵列结构仍有其优点,这就是简单易行,有利于扩展阵列规模,且对于相当一部分算法具有较高的效率。

(3) ILLIAC IV 硬件设备量过于庞大,造价很高,且使系统工作可靠性受到不利影响。这固然与六十年代的电子器件工艺水平较低有关,但从系统结构上以设备换取速度的设计思想也起了主要作用。

(4) ILLIAC IV 的运行方式大多是批处理,专用性又很强,因此不要求过于先进的系统软件。虽然在系统投入运行以来的十年中,充实了各种系统和应用软件,但仍有大量问题需要研究解决。

在并行处理机发展过程中怎样看待和解决上述问题和困难呢?概括说来,有两条途径。一是彻底摆脱阵列结构、分布式并行存贮器等带来的限制,转向完全新型的系統,其中可以采用较强功能的互连网络和集中式共享存贮器,但有必要减少处理单元数目。美国 Burroughs 公司的 BSP 科学处理机是这类新系统的代表。二是基本继承和发扬 ILLIAC IV 的阵列结构特性,但引入若干重大改进,以适应更大阵列规模的需要。在这个方向上发展的典型例子是:英国 ICL 公司的 DAP 分布式阵列处理机;美国 Goodyear 宇航公司的 MPP 巨型并行处理机;西德的 EGPA 通用阵列处理机等。下面将以 MPP 和 BSP 为例,分别介绍一些大致的情况。

2.4-2 MPP 位平面阵列处理机

MPP 是美国 Goodyear 宇航公司专门设计用于遥感卫星图象处理的巨型并行处理机。它对 ILLIAC IV 阵列结构的重大改进就是采用位平面结构。它的中心思想是:每一个处理单元只包含一位的硬件设备,单元内部的算术逻辑运算串行地完成。这不但节省了设备,还使处理单元之间的信息通路宽度大为减小。与字长 64 位的 ILLIAC IV 阵列相比,硬件设备量可以减少几十倍;换句话说,用同样的硬件设备至少可以扩大阵列规模几十倍。此外,减小处理单元之间的信息通路宽度还有利于增加它们的互连模式,提高阵列结构的灵活性。这主

要是指：处于阵列相对边缘上的那些处理单元之间可以实现多种不同的连接。

MPP并行处理机的原理框图如图 7.13 所示。它的阵列部件 ARU 由 128×128 个处理单元 PE 组成，每一个 PE 附有一个 1024 位本地随机存储器。同时，每一个 PE 的算术逻辑操作是由一个串行加法器和一个逻辑部件来完成的，利用一个可变字长的移位寄存器将操作数一位一位地送入加法器。根据信息处理的需要，字长可以设置为 4、8、12、16、20、24、28 或 32 中的任一种。基本时钟频率是 10MC，循环周期为 100ns。

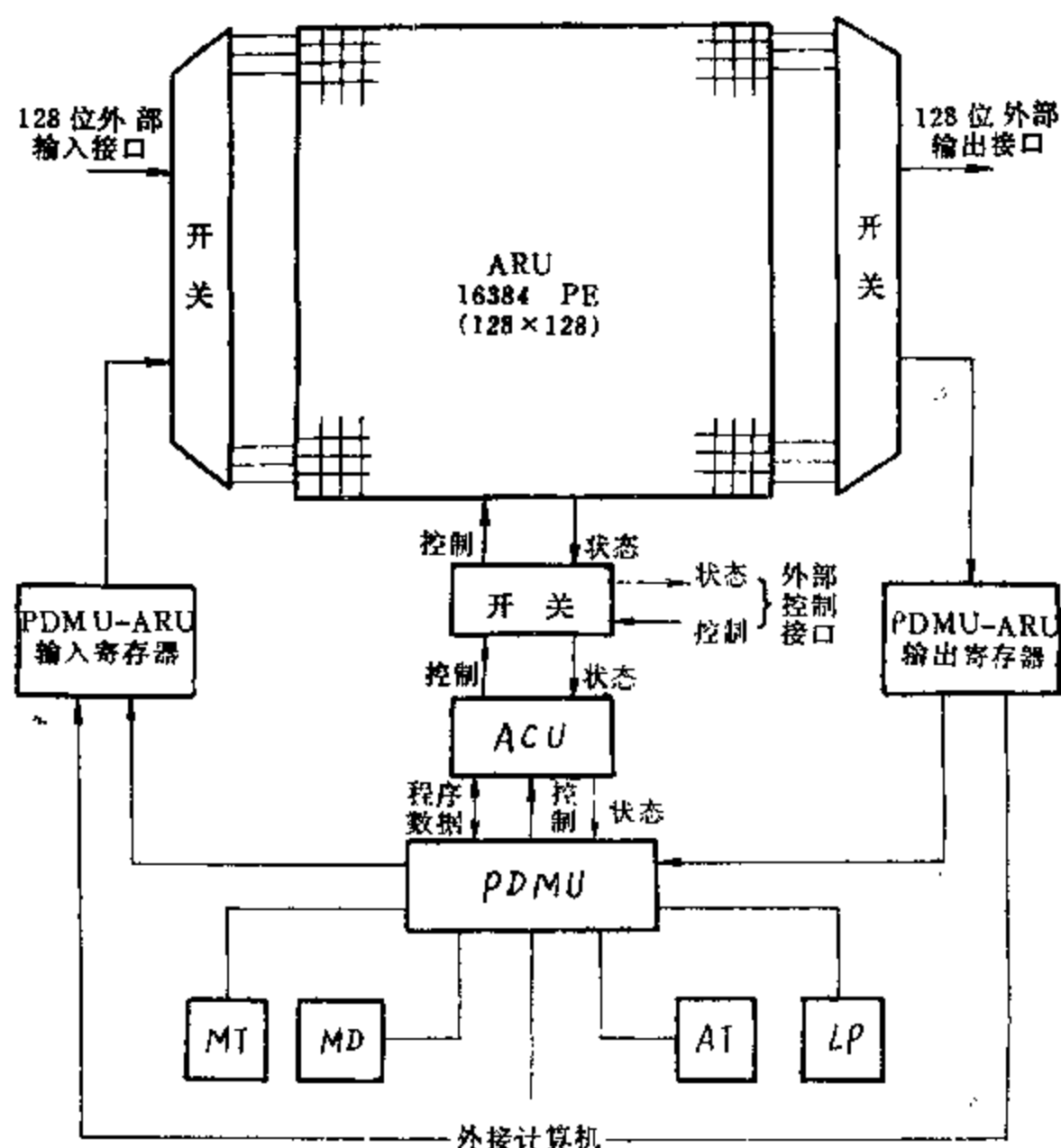


图 7.13 MPP 并行处理机原理框图

MPP 阵列保持了 ILLIAC IV 的互连方式，即每一个 PE 只与其东、南、西、北四个近邻相连接，而阵列四个边缘的连接方式可由程序员加以改变，在五种不同的阵列拓扑 (topology)，即相互连接模式中任选一种：平面、水平圆柱、垂直圆柱、开螺线或闭螺线。这一改进措施使在某些应用情形下的连接路径大为缩短，并增强阵列结构的灵活性。

阵列 ARU 的左、右两边各有一排 128 位的开关，各通过 128 根并行线与阵列相连。借助它们使阵列从左端输入数据和向右端输出数据。输入和输出既可以单独进行，也可以同时进行，并和处理单元的操作时间相重迭。每一时钟周期可以并行传送一列数据，使最大传送速率等于 $128 \text{ 位} / 100 \text{ 毫微秒} = 1.28 \times 10^6 \text{ 位/秒}$ 。设置这两排开关，使阵列既可以与外部接口交换数据，又可以与内部寄存器交换数据。

整个系统的控制任务是由程序及数据管理部件 PDMU 来承担的。它实际上是一台标准的 PDP-11/34 小型计算机，带有磁带机 MT、磁盘机 MD、宽行打印机 LP 和字符显示终端 AT

等外围设备。PDMU可与三个方面相接口：一是PDMU-ARU输入寄存器和PDMU-ARU输出寄存器，经过上述两排开关与阵列ARU相连，用以交换数据；二是阵列控制器ACU，经过开关与阵列ARU相连，用以向阵列的全部处理单元播送控制信号、存贮地址和从阵列取得状态码（开关的设置使这些信息也可以来自和送往外部控制接口）；三是MPP的外接计算机，它的数据目的地和来源也可以分别是PDMU-ARU输入/输出寄存器。PDMU的作用是管理MPP各部件间的信息流动，包括控制所有的电子开关，向ACU加载程序、执行系统的测试和诊断程序、提供程序开发能力等。

图7.13的系统组成表示出MPP不但在阵列结构上具有灵活性，而且在运行方式上也具有灵活性。它可有三种运行方式：独立方式、联机方式和高速数据方式。在独立方式下，所有程序开发、执行、测试和调整都在MPP系统内部完成，由PDMU的磁盘和磁带供给数据，由PDMU字符终端上操作员的命令控制。而联机方式则是有外接计算机参加，送入数据、常数、程序和作业要求，并接收结果数据和系统或程序的状态信息；外部计算机与MPP之间的数据传输率是 6×10^6 字节/秒。在高速数据方式下，数据的输入和输出是通过128位外部接口来进行的，由MPP控制，数据传输率达到 160×10^6 字节/秒。

2.4-3 BSP 科学处理机

BSP是美国Burroughs公司和依里诺大学计算机科学系合作设计的新型并行处理机。主要针对科学计算任务，这就是它的名称的由来。最高处理速度是每秒五千万次浮点运算。从它的性能和设计先进性来看，被认为是第二代并行处理机。

首先，BSP系统采取了全面的并行化措施。它获得高速并不是依靠提高时钟频率，而是依靠并行性。它的时钟频率在巨型计算机中是比较低的：处理器和存贮器的周期都采用160ns，但是依靠重复设置16个处理单元，仍能获得与CRAY-I流水线处理机相当的向量处理速度。

BSP还采用共享的多体存贮器，通过高速互连网络与16个处理单元相连接。为了对于多维向量元素的各种组合（例如二维向量的行、列、对角线、反对角线等）都能实现存贮器的无冲突并行访问，它采用了特殊的存贮器系统结构，即选用与16最靠近的质数(17)个存贮器模块，并利用特殊的地址映象算法实现所需数组元素的读写操作。

这样，BSP的16个处理单元、17个存贮器模块和2套互连网络（亦称对准网络）组合在一起，就形成了一条五级的数据流水线，使连续几条向量指令能在时间上重迭起来执行。其结构示意图如图7.14所示。五级的功能作用依次是：

- (1) 由17个存贮块并行读出16个操作数；
- (2) 经对准网络NW1将16个操作数重新排列成16个处理单元所需要的次序；
- (3) 将排列好的16个操作数送到并行处理单元完成操作；
- (4) 所得的16个结果经对准网络NW2重新排列成17个存贮器模块所需要的次序；
- (5) 写入存贮器。

流水线由统一的指令译码和控制部件进行控制。

这种流水线结构是很新颖的，对提高系统处理效率起着很大的作用。第一，它有效地实现了处理单元、存贮器和互连网络在时间上的重迭工作，在理想情况下能取得频带的完全匹配。第二，它把任意长度(>16)的向量按16个分量的标准长度分为若干段，依次在时间上

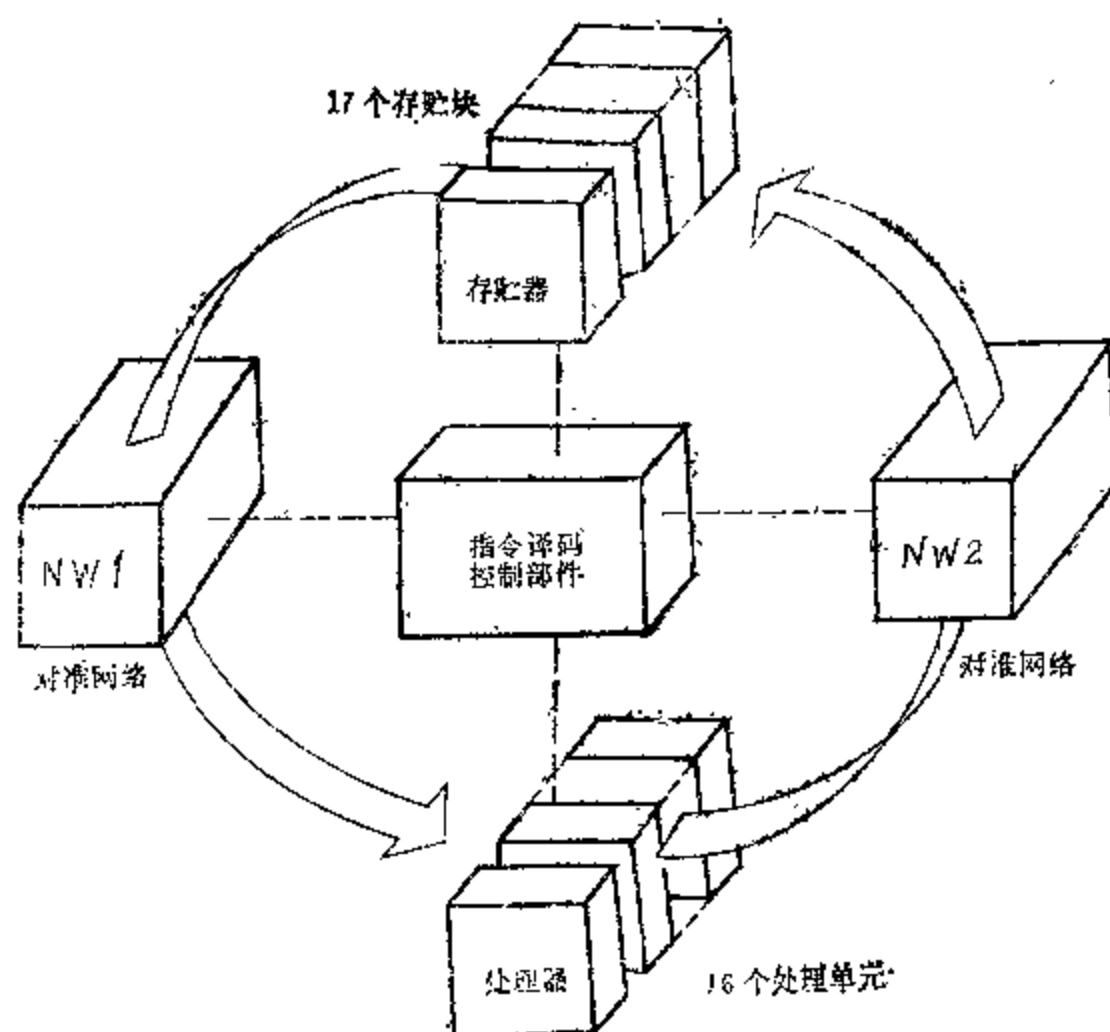


图 7.14 BSP 五级数据流水线结构示意图

重迭起来进行处理。第三，当一条向量指令的最后一段开始处理时，下一条向量指令的最初一段即可进入流水线，实现不同向量指令的重迭执行。这个五级数据流水线的前面还要加上控制流水线，它使指令的建立过程也可与后面的执行过程在时间上重迭起来，这样来减小指令建立时间的开销。

象 BSP 这样把资源重复和时间重迭两种并行性结合起来，有助于克服一般 SIMD 计算机在向量处理上的局限性。例如说，有一些向量处理机的性能随向量长度增加才能增强，另一些向量处理机的性能取决于向量长度是否与高速寄存器组的容量或处理单元的数目相匹配。BSP 区别于这些机器的优点是：它的处理效率较少受向量长度和指令建立时间的影响，因此对短向量的处理同样有利。

BSP 还继承和发展了 ILLIAC IV 中把标量指令和向量指令重迭起来的作法。

此外，对 BSP 的性能有重要作用的还应提到它的高水平指令系统。它使用与 FORTRAN 程序结构十分协调的数组指令形式。高级机器语言由于与源程序较好配合，能够简化编译程序和控制部件的设计，并显著改善系统性能；但是数组指令的建立时间加长往往又是一个难于解决的问题。BSP 的流水线结构有助于它成功地采用高级数组指令。这些指令可以表示一个完整的数组赋值语句，其右边部分可以包含多至 5 个向量操作数和任何操作组合，而且操作数可以是任意长度的一维或二维数组。指令还可以是多种特殊表达式，如向量和积、向量内积、一阶线性递归、下标表读取等。

高级数组指令的采用使上述五级数据流水线随时都能充满而很少中断，充分发挥了流水线的效能。而在出现错误需要中断或复执时，流水线又很容易在任何一个时钟节拍上停下来转入错误处理。

BSP 还有一个高效能的 FORTRAN 编译程序，具有很强的向量化功能，对程序中隐

含并行性能保证较高的识别率。它不但适用于标准的 ANSI FORTRAN 语言, 还能针对向量扩充的 FORTRAN 语言。向量化程序不但能够处理明显的数组操作, 还能处理线性递归、循环内部的条件分支等进程, 产生显著的加速效果。

BSP 科学处理机系统组成框图如图 7.15 所示。它由系统管理计算机 B-7700/B-8700 和 BSP 科学处理机两大部分组成。前者可视为后者的前端机, 担负 BSP 程序编译、任务调度、数据通信和外围设备管理等任务。大多数 BSP 作业调度和操作系统活动都是在系统管理计算机上完成的。BSP 科学处理机本身又可分为三部分: 一是并行处理机, 包含 16 个算术单元和容量为 524K 字的并行存贮器; 二是控制处理机, 带有容量为 256K 字的指令/控制存贮器; 三是容量为 4M 字的文件存贮器。

控制处理机主要用作并行处理机和文件存贮器系统的控制器。它还包含一个运算单元, 具有整数运算和变址操作功能, 用来执行用户程序中某些串行或标量部分。此外, 控制处理机还有一个维护诊断接口与系统管理机相连。

文件存贮器是一个高速大容量外存贮器, 是置于 BSP 科学处理机直接控制之下的唯一

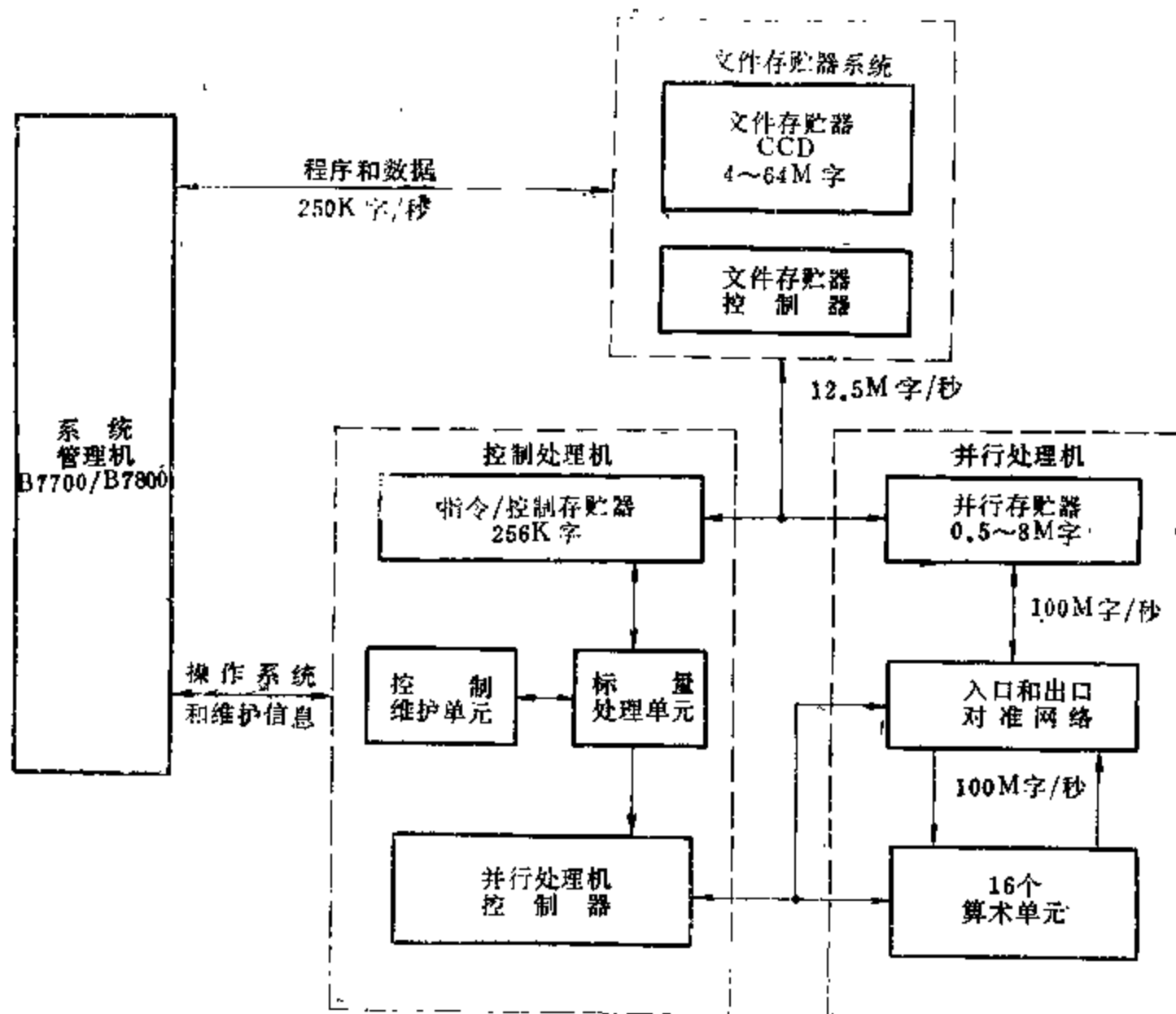


图 7.15 BSP 科学处理机系统组成框图

外围设备。它和并行存贮器紧密耦合, 可视为该存贮器的直接扩充。它还用来存放 BSP 的计算任务文件, 由系统管理机加载, 再由这里安排队次序转送指令/控制存贮器, 以便执行。

2.5 SIMD 计算机的互连网络

2.5-1 互连网络问题的重要性

在多计算机系统中,无论是处理机之间,还是处理器与存储器之间,都要通过互连网络实现信息交换。决定互连网络性能价格比的诸因素中,主要的是结构复杂性(反映成本)和通信频带及结构灵活性(反映性能)。在大规模集成电路和微处理器飞速发展的今天,建造多达 $2^{14} \sim 2^{16}$ 个处理部件的并行处理机已是现实可能的了,但是要把这么多处理部件连接起来,而又具有足够的灵活性,却是一个困难的问题。当然,要求任意两个处理部件之间都有直接连接,对于这么庞大的系统来说,显然是不现实的,所以不得不暂时满足于象 ILLIAC IV 这样仅仅在处理部件之间实现有限的连接。这种被称为闭合螺线阵列的方案,经证明仍能适应许多种常见算法,已如前述。但是能不能找到更灵活的互连网络结构,扩大它对算法的适应范围,这是多机系统结构研究中的一个关键问题。

目前已有许多种互连网络被提出来,有些并已得到实际的采用。许多文献对各种互连网络进行了统一的综合的分析研究工作。在美国还为这一课题召开了专门的学术讨论会,足见互连网络理论和技术发展的重要性。

下面先介绍单级互连网络,进而介绍多级互连网络,它们都是针对处理器数目很多的并行处理系统而设计的,统称为 SIMD 互连网络。但是,这不等于说,它们就只能在 SIMD 计算机中应用。它们的目标是要以较少的步数实现任意两个处理器之间的连接,这与 MIMD 计算机的要求是一致的。只不过 MIMD 计算机目前大多是处理器数目较少的系统,它们采用另外一些传统的结构方式,这待下节再讨论。

2.5-2 单级互连网络

单级互连网络可以看成是一级开关,按照不同的拓扑关系,把 N 个入端和 N 个出端连接起来。下面将要看到,ILLIAC IV 阵列就是单级互连网络的一种,并且是一种普遍结构形式的一个特例。

一般说,单级互连网络可以有多种形式,其中立方体单级网络、PM2I 单级网络和混洗交换(Shuffle-exchange)单级网络是最基本的三种。

为了反映不同互连网络的连接特性,每一种互连网络可用一组互连函数来定义。如果把互连网络的 N 个入端和 N 个出端($N=2^n$)各自用整数 $0, 1, \dots, N-1$ 代表,则互连函数乃是表示互连的出端号和入端号的一一对应关系。令互连函数为 f ,则它的作用是:对于所有的 $0 \leq j \leq N-1$,同时存在入端 j 连至出端 $f(j)$ 的对应关系。由于在建立处理器之间的互连时,入端和出端实际上是属于同一数组,所以互连函数也反映了一种“排列”关系,这种关系可用“循环”来表示。例如,若把互连函数 f 表示为

$$(j_0 \ j_1 \ j_2 \ \dots \ j_{x-1} \ j_x),$$

则代表对应关系

$$\begin{aligned} f(j_0) &= j_1, f(j_1) = j_2, \dots, f(j_{x-1}) = j_x, \\ f(j_x) &= j_0. \end{aligned} \quad (7.2-5)$$

$x+1$ 称为该循环的长度。

下面就分别讨论三种基本的单级互连网络。

立方体单级网络的名称来源于图 7.16 所示的立方体结构，它表示一个三维的情形，立方体的每一个顶点代表一个处理器，用 zyx 三位二进制代码予以标号， z 、 y 和 x 可以各等于 0 或 1。

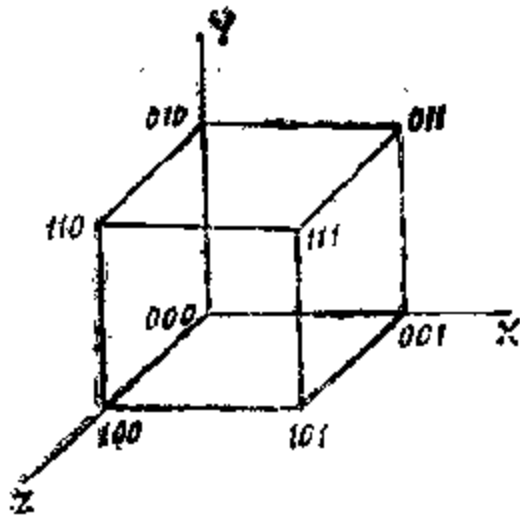


图 7.16 三维立方体结构

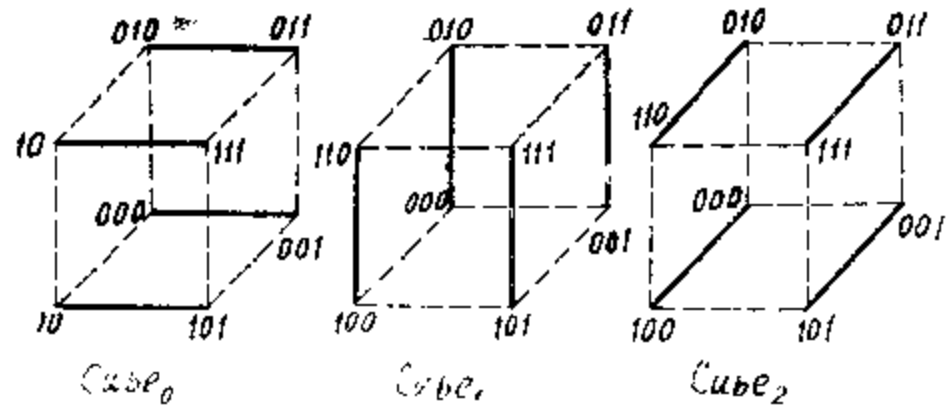


图 7.17 立方体单级网络连接图

三维的立方体单级网络有三种互连函数： $Cube_0$ 、 $Cube_1$ 和 $Cube_2$ ，其连接方式如图 7.17 中实线所示。 $Cube$ 函数下标数字 0 (或 1, 2) 表示相连的入端和出端的二进制标号只在最低 0 位 (或 1, 2 位) 上有差别，即“0”和“1”码互反，而其余各位代码均相同。推广到 n 维的情形，立方体单级网络共有 $n = \log_2 N$ 种互连函数，即

$$Cube_i(P_{n-1} \dots P_i \dots P_1 P_0) = P_{n-1} \dots \overline{P_i} \dots P_1 P_0 \quad (7.2-6)$$

式中 P_i 为入端的二进制标号第 i 位的代码，且 $0 \leq i \leq n-1$ 。

PM2I 单级网络是“加减 2^i ” (Plus-Minus 2^i) 单级网络的简称，共包含 $2n$ 个互连函数，即

$$PM2_{+i}(j) = j + 2^i \bmod N \quad (7.2-7)$$

$$PM2_{-i}(j) = j - 2^i \bmod N$$

式中 $0 \leq j \leq N-1$, $0 \leq i \leq n-1$, $n = \log_2 N$ 。例如，对于 $N=8$ ，各互连循环为

$$PM2_{+0}: (0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)$$

$$PM2_{-0}: (7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \ 0)$$

$$PM2_{+1}: (0 \ 2 \ 4 \ 6) \ (1 \ 3 \ 5 \ 7)$$

$$PM2_{-1}: (6 \ 4 \ 2 \ 0) \ (7 \ 5 \ 3 \ 1)$$

$$PM2_{+2}: (0 \ 4) \ (1 \ 5) \ (2 \ 6) \ (3 \ 7)$$

普遍而言， $PM2_{+(n-1)} = PM2_{-(n-1)}$ 。

PM2I 互连网络的连接图示于图 7.18，其中只画出 n 种互连函数的情况，对于其余 $PM2_{-i}$, $0 \leq i \leq n-2$ 等 $n-1$ 种互连函数，连接的箭头正好相反。ILLIAC IV 阵列就是 PM2I 互连网络的一个特例，它包含 $PM2_{+0}$ 和 $PM2_{+n/2}$ 等四个互连函数。

混洗交换网络^[22]包含二个互连函数，一个是全混 (perfect shuffle)，另一个是交换 (exchange)。图 7.19 表示 8 个处理器之间的全混连接，可以看出其连接规律是把全部按标号次序排列的处理器从当中分为数目相等的两半，前半和后半在连至出端时正好一一隔开。正好象人们在洗 (shuffle) 扑克牌时，先把整付牌分为两半，然后洗牌达到理想

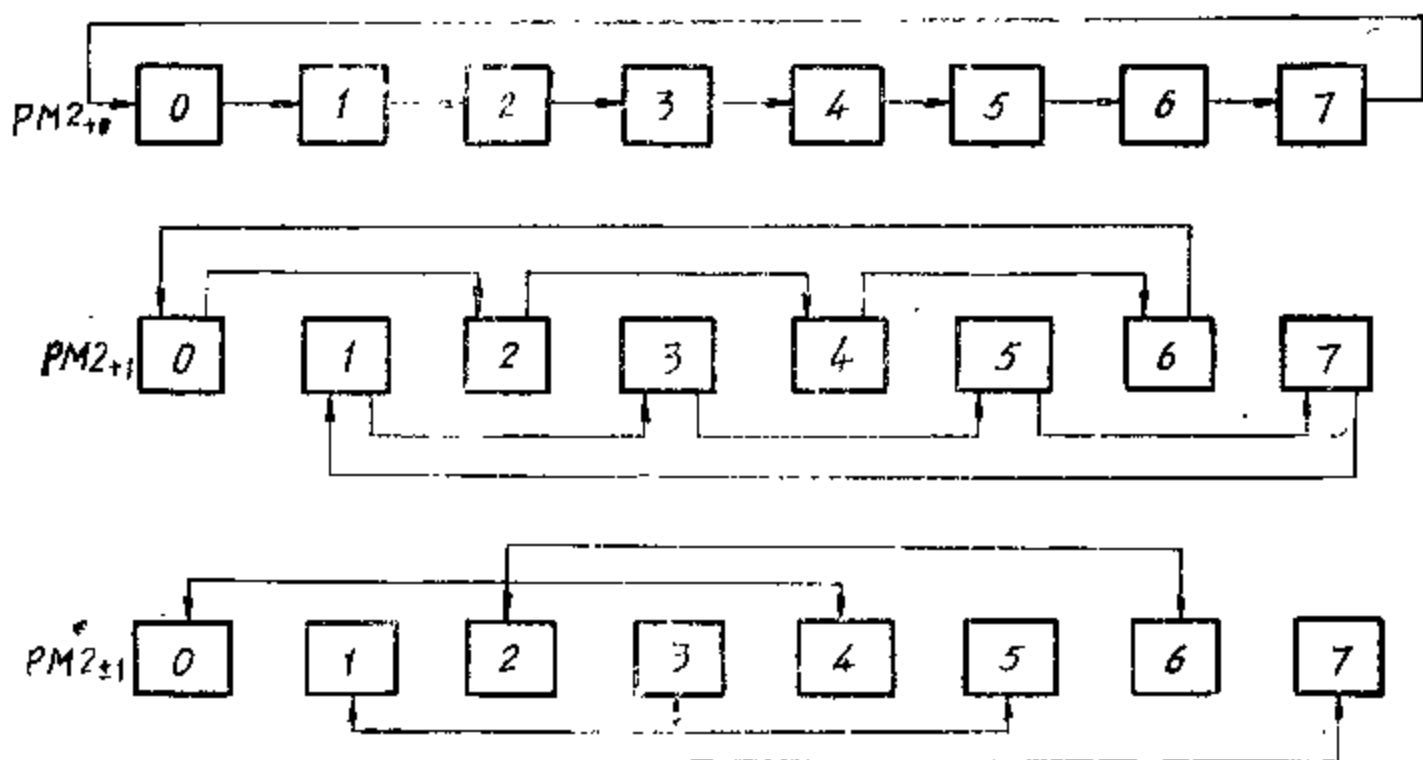


图 7.18 PM2I 互连网络连接图

的“全混”状态一样。这也是“混洗”这个名词的由来。用互连函数表示为：

$$\text{shuffle}(P_{n-1}P_{n-2}\cdots P_1P_0) = P_{n-2}\cdots P_1P_0P_{n-1} \quad (7.2-8)$$

式中， $n = \log_2 N$ ， $P_{n-1}P_{n-2}\cdots P_1P_0$ 为入端标号的二进制代码。

注意，shuffle 函数不是可逆函数。如果把出端当作入端，入端当作出端，则原网络变为另一个互连网络，称逆互连网络，这相当于图 7.19 中把箭头逆转的情况。

shuffle 函数还有一个重要特性。如果把它再作一次 shuffle 函数变换，则得到的是一组新的代码，即 $P_{n-3}\cdots P_0P_{n-1}P_{n-2}$ 。这样，每全混一次，新的最高位就移至最低位一次，当全混总次数为 n 时，全部 N 个处理器便又恢复到最初的排列次序。但在这多次全混过程中，每个处理器都遇到了与其它多个处理器连接的机会，这正是 shuffle 函数的用处所在。

由于单纯的全混互连网络不能实现编号全“0”的处理器以及编号全“1”的处理器与其它处理器的任何连接，所以还必须增加交换互连函数，它就是 Cube。于是得到全混交换单级网络，其 $N=8$ 的连接图如图 7.20 所示。其中实线表示交换，虚线表示全混。从图中也可看到，全混 $\log_2 8 = 3$ 次回到原位的情况。

全混连接与立方体连接之间存在着很有意义的对应关系，只需将图 7.17 与图 7.20 对照比较就能看出来。第一次全混后，入端编号为 (0,4)，(1,5)，(2,6)，(3,7) 的各处理器对正好在出端相邻，这就进入 Cube_2 的循环；接着，如果第二次全混，则出端相邻的是 Cube_1 的处理器对；第三次全混后，出现 Cube_0 的排列，便又恢复原位。全混连接这一性质使它便于构成多级连接，并与立方体多级连接具有相似的关系，这将在下面再讨论。

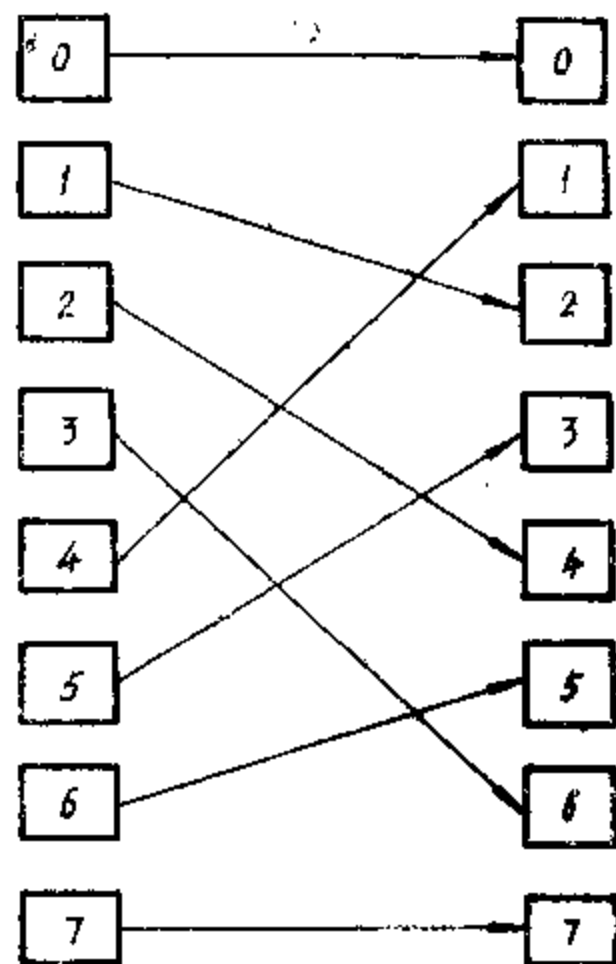


图 7.19 8 个处理器的全混连接

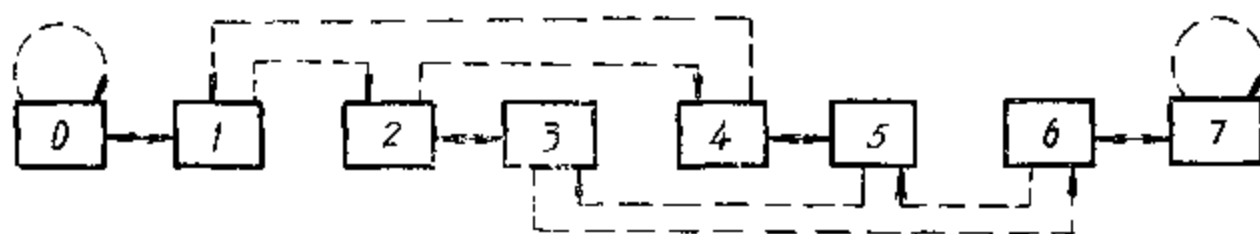


图 7.20 $N=8$ 时全混交换互连网络连接图

单级互连网络不论是哪一种，都可以表示为一普遍模型，如图 7.21 所示。其中 IS 代表入端选择器，OS 代表出端选择器，二者配合能实现 N 个入端和 N 个出端之间的各种连接。

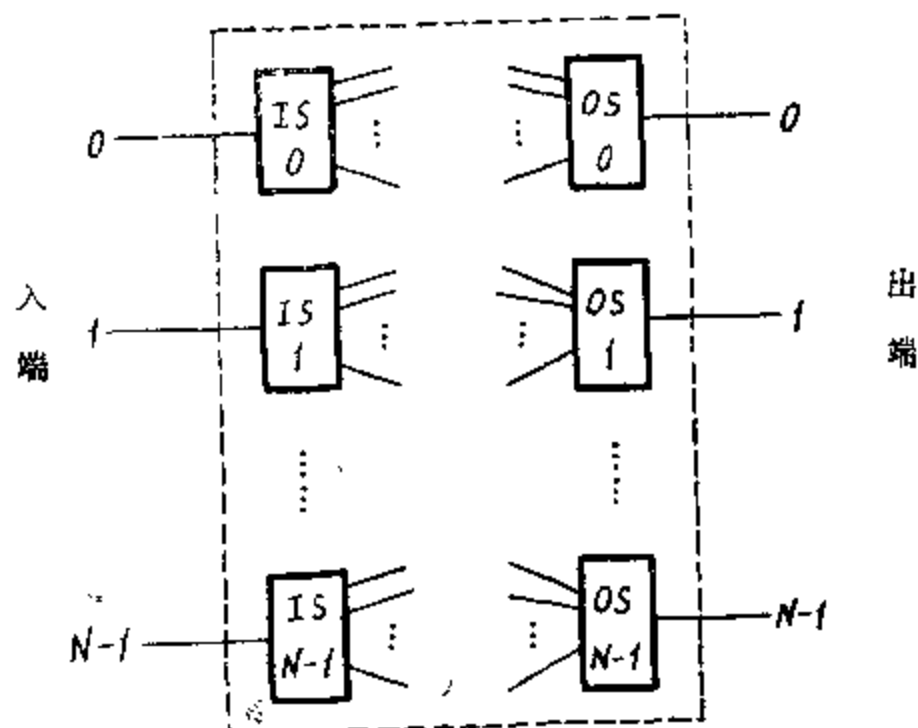


图 7.21 单级互连网络的模型

2.5-3 循环互连网络和多级互连网络

单级互连网络有两种可能的使用方式：一种是同一套单级互连网络循环使用，组成循环互连网络；另一种是多套单级互连网络串连，组成多级互连网络。

循环互连网络的模型如图 7.22 所示。入端传送寄存器 DTR_i 和出端传送寄存器 DTR 。除了与处理器 $PE_0 \sim PE_{N-1}$ 相连，分别接收和送出数据外，还可以通过多路开关 MUX 在不同的循环中向单级互连网络送入 DTR_i 的数据，经过单级互连网络转接后再送回 DTR_0 ，以便重复利用。这种循环互连网络与多级互连网络相比，节省了重复的设备，但加长了通过时间。

多级互连网络可以利用上述各种单级互连网络进行不同的组合，因此也可以有多种。其中多级立方体网络、多级混洗交换网络和多级 PM2I 网络是最基本的。

一般说来，不同的多级互连网络在下列三个参量上互相区别：交换开关、拓扑结构和控

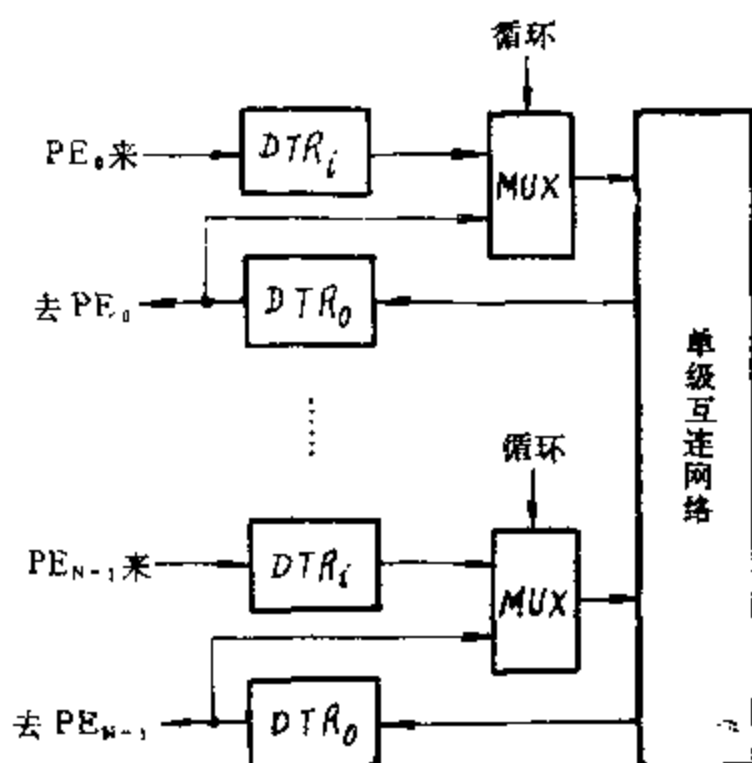


图 7.22 循环互连网络组成框图

制方式。

交换开关是具有二入端和二出端的交换单元，用作各种多级互连网络的基本构件。不论入端或出端，令居于上方的都用 i 表示，居于下方的都用 j 表示，则可以定义下列四种开关状态或连接方式：(1) 直连—— i 入连 i 出， j 入连 j 出；(2) 交换—— i 入连 j 出， j 入连 i 出；(3) 上播—— i 入连 i 出和 j 出， j 入悬空；(4) 下播—— j 入连 i 出和 j 出， i 入悬空。只具有前二种功能的称二功能交换单元，具有全部四种功能的称四功能交换单元。两个入端同时连到一个出端的情形是不允许的，称为冲突状态。

拓扑结构是指各级之间出端和入端相互连接的模式。上述单级互连网络的那些连接模式都可以被利用来进行不同的组合，构成多种不同的多级互连网络。

控制方式是对各个交换开关进行控制的方式，它可以有三种：(1) 级控制——同一级的所有开关只用同一信号控制，同时只能处于同一状态；(2) 单元控制——每一单元都有自己的控制信号，可各自处于不同的状态；(3) 部分级控制——用 $i+1$ 个信号控制第 i 级，此处， $0 \leq i \leq n-1$ ， n 为级数。规定控制信号用写在交换单元框内的文字代表，如果不同框内写的是同一字母，则表示所用的控制信号相同。

利用上述三个参量，可以来描述各种多级互连网络的结构。

多级立方体网络有 STARAN 网络^[20]、间接二进制 n 方体网络 (Indirect binary n -cube network)^[21] 等，其普遍结构如图 7.23 所示。其共同特点是：第 i 级 ($0 \leq i \leq n-1$) 交换单元处于交换状态时，实现的是 $Cube_i$ 互连函数，且都采用二功能交换单元。二者的差别仅在于控制方式上：STARAN 网络用级控制[称交换网络 (Flip network)]和部分级控制[称移数网络 (Shift network)]，而间接二进制 n 方体网络用单元控制，因此，后者具有更大的连接灵活性。

STARAN 网络是多级互连网络中已经获得成功应用的一种，它也以在其中获得实际应用的巨型相联处理机 STARAN 而得名。它和该系统的多维相联存贮器一起，是系统的重要组成部分，用来对 256 位的数据进行各种处理和变换。当它用作交换网络时，实现的是交换

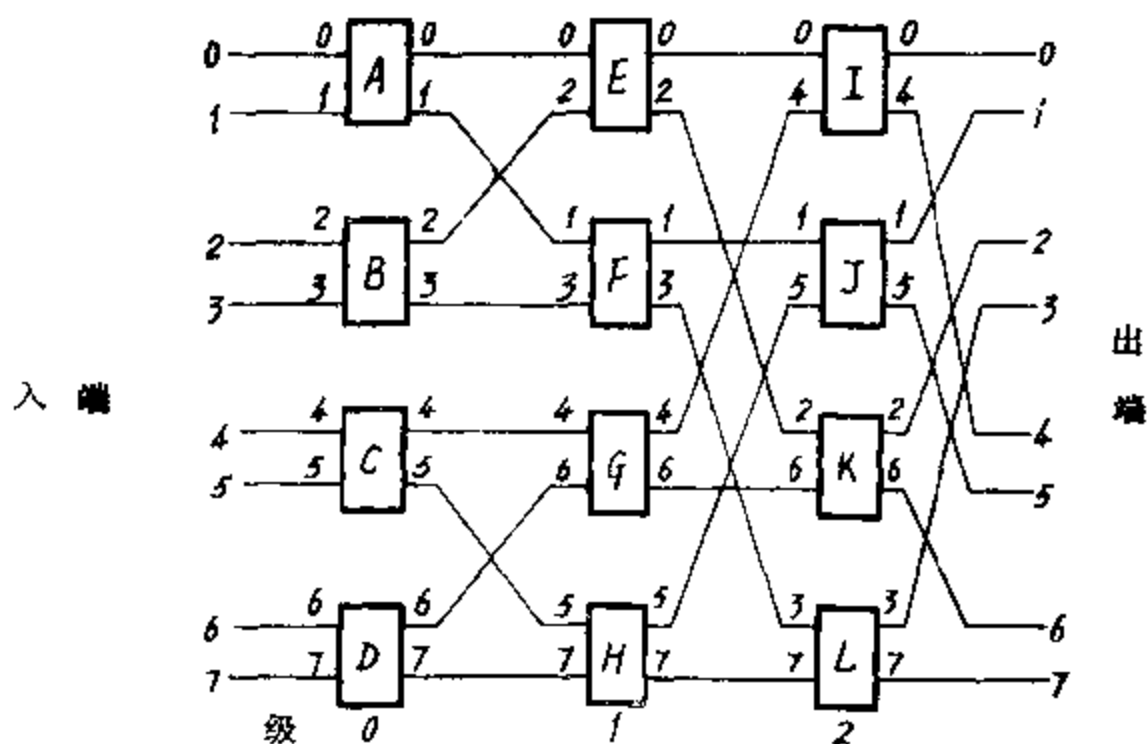


图 7.23 $N=8$ 多级立方体互连网络

函数。所谓交换 (Flip) 函数，根据文献[23]中给出的定义，就是将一组元素头尾对称地进

行交换。如果一组包含 2^i 个元素，则第 k 个元素与第 $(2^i - k + 1)$ 个元素相交换。这时，STARAN 网络采用级控制，当第 i 级 $(0 \leq i \leq n-1)$ 控制信号为“1”时，该级所有的交换单元同时处于交换状态，出端编号是入端编号在第 i 位代码上变反；当该信号为“0”时，则相应交换单元同时处于直连状态，代码不变。如果只有第 i 级控制信号为“1”，其余各级控制信号为“0”，则实现的互连函数恰好是 $Cube_i$ 。表 7.2 列出了三级交换网络的控制结果。

表7.2 三级 STARAN 交换网络的控制信号与互连循环的关系

控 制 信 号			互 连 循 环	实 现 函 数
I,J,K,L	E,F,G,H	A,B,C,D		
0	0	0	(0)(1)(2)(3)(4)(5)(6)(7)	恒等
0	0	1	(0 1)(2 3)(4 5)(6 7)	4 组 2 元交换
0	1	0	(0 2)(1 3)(4 6)(5 7)	4 组 2 元交换 + 2 组 4 元交换
0	1	1	(0 3)(1 2)(4 7)(5 6)	2 组 4 元交换
1	0	0	(0 4)(1 5)(2 6)(3 7)	2 组 4 元交换 + 1 组 8 元交换
1	0	1	(0 5)(1 4)(2 7)(3 6)	4组2元交换 + 2组4元交换 + 1组8元交换
1	1	0	(0 6)(1 7)(2 4)(3 5)	4 组 2 元交换 + 1 组 8 元交换
1	1	1	(0 7)(1 6)(2 5)(3 4)	1 组 8 元交换

从表 7.2 可以看出，控制信号为 111 时，实现的是全交换，又称镜象交换，其变换图象如下：

入端排列 0 1 2 3 4 5 6 7
出端排列 7 6 5 4 3 2 1 0

普遍言之，控制信号最高位为“1”时，其变换图象都可从控制信号为其反码的变换图象再经一次镜象交换得到。表 7.2 中的后 4 组和前 4 组之间就说明了这一关系；而前 4 组中如果只看控制信号的低二位，例如控制信号 011 和 000 之间以及 010 和 001 之间，也存在 4 元的镜象交换关系。

当 STARAN 网络用作移数网络时，采用部分级控制，控制信号分组和控制结果列在表 7.3 中。

表7.3 STARAN 移数网络控制信号与互连循环的关系

控 制 信 号						互 连 循 环	实 现 函 数
K,L	J	I	F,H	E,G	A,B,C,D		
0	0	1	0	1	1	(0 1 2 3 4 5 6 7)	移 1 mod 8
0	1	1	1	1	0	(0 2 4 6)(1 3 5 7)	移 2 mod 8
1	1	1	0	0	0	(0 4)(1 5)(2 6)(3 7)	移 4 mod 8
0	0	0	0	1	1	(0 1 2 3)(4 5 6 7)	移 1 mod 4
0	0	0	1	1	0	(0 2)(1 3)(4 6)(5 7)	移 2 mod 4
0	0	0	0	0	1	(0 1)(2 3)(4 5)(6 7)	移 1 mod 2
0	0	0	0	0	0	(0)(1)(2)(3)(4)(5)(6)(7)	恒 等

STARAN 网络用在 STARAN 相联处理机的多维访问存储器与处理部件之间对存储器中杂错存放的数据在读出后和写入前进行重新排列,以适应处理部件对数据正常位序的需要。利用 STARAN 网络的交换和移数这两种基本功能,加上对数据位进行适当屏蔽,还能实现全混、展开、压缩等多种数据变换函数。

多级混洗交换网络又称 omega 网络,如图 7.24 所示。它由 n 级相同的网络组成,每一级都包含一个全混拓扑和随后一系列 2^{n-1} 个四功能交换单元,采用单元控制方式。比较图 7.23 和图 7.24,可以发现一个有意思的情况,即 omega 网络中各级的编号次序与多级立方体网络正好相反。这并不是偶然的,因为它保证了在编号相同的级内所有交换单元入端和出端的配对情况恰好相同。这说明,如果把 omega 网络的入端和出端位置对调,它就等同于间接二进制 n 方体网络。因此可以断定:omega 网络与间接二进制 n 方体网络之间只存在两点差别:(1)前者的数据流向是级号 $n-1, n-2, \dots, 1, 0$,而后者的数据流向是级号 $0, 1, \dots, n-1$,正好相反;(2)前者用四功能交换单元,后者用二功能交换单元。

如果暂时假定 omega 网络也采用二功能交换单元,那么单是数据入出流向相反对互连排列有何影响呢?当然,差别是明显的,例如, n 方体网络能实现 5 到 0, 7 到 1 的连接,但不能实现 0 到 5, 1 到 7 的连接;而 omega 网络则正好相反,5 到 0 和 7 到 1 不行,0 到 5 和 1 到 7 却可以。有一种方法把二者统一起来,就是入端和出端重新编号。仍比较图 7.23 与图 7.24,可以发现,如果把编号 $P_{n-1} P_{n-2} \dots P_1 P_0$ 和 $P_0 P_1 \dots P_{n-2} P_{n-1}$ 互换,例如对于 $n=3$,就是把 1, 4 互换, 3, 6 互换,那么两张图上所有交换单元上的处理器对就都变成一样的了。于是表 7.2 和表 7.3 中所列 STARAN 网络的互连函数,在按上述规则对处理机重新编号后,便都适用于 omega 网络,而且所有在前者运行的 SIMD 程序都能向上兼容。当然,由于 omega 网络采用四功能交换单元,因此允许同时实现一个处理器与多个处理器的连接,这是多级立方体网络不可能办到的。例如,只需将图 7.24 中交换单元 E, F 置为下播状态, C, I, J, K, L 置为上播状态,就能一次实现入端 2 与全部 8 个出端的连接。

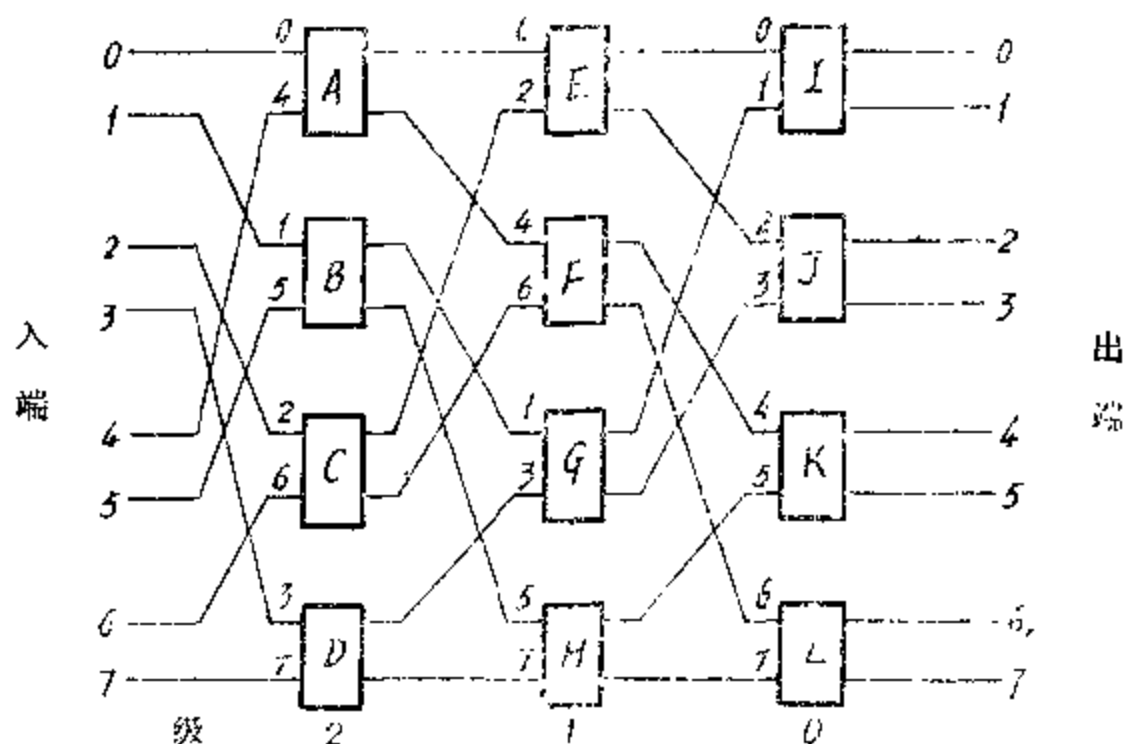


图 7.24 $N=8$ 多级混洗交换网络

多级 PM2I 网络又称为数据变换网络。对于 $N=8$, 其结构如图 7.25 所示。它包含 n 级单元间连接,每一级都是把两列各 $N=2^n$ 个单元按 PM2I 拓扑相互连接起来。从第 i 级

($0 \leq i \leq n-1$) 来说, 每一个入单元 j ($0 \leq j \leq N-1$) 都有三根连接线分别通往出单元 j , $j+2^i \bmod N$ 和 $j-2^i \bmod N$, 在图 7.25 中它们分别用点线、实线和虚线表示。控制这三组连接线的信号分别称为平控 H 、下控 D 和上控 U 。这些控制信号的分组情况为: 对于第 i 级, H_i^j , D_i^j , U_i^j 控制第 i 位为“0”的那些入单元, 而 H_i^j , D_i^j , U_i^j 控制第 i 位为“1”的那些入单元。还有一种网络称为强化数据变换网络, 缩写为 ADM, 是采用单元控制, 即每一单元都有自己的控制信号 H , D 和 U 。

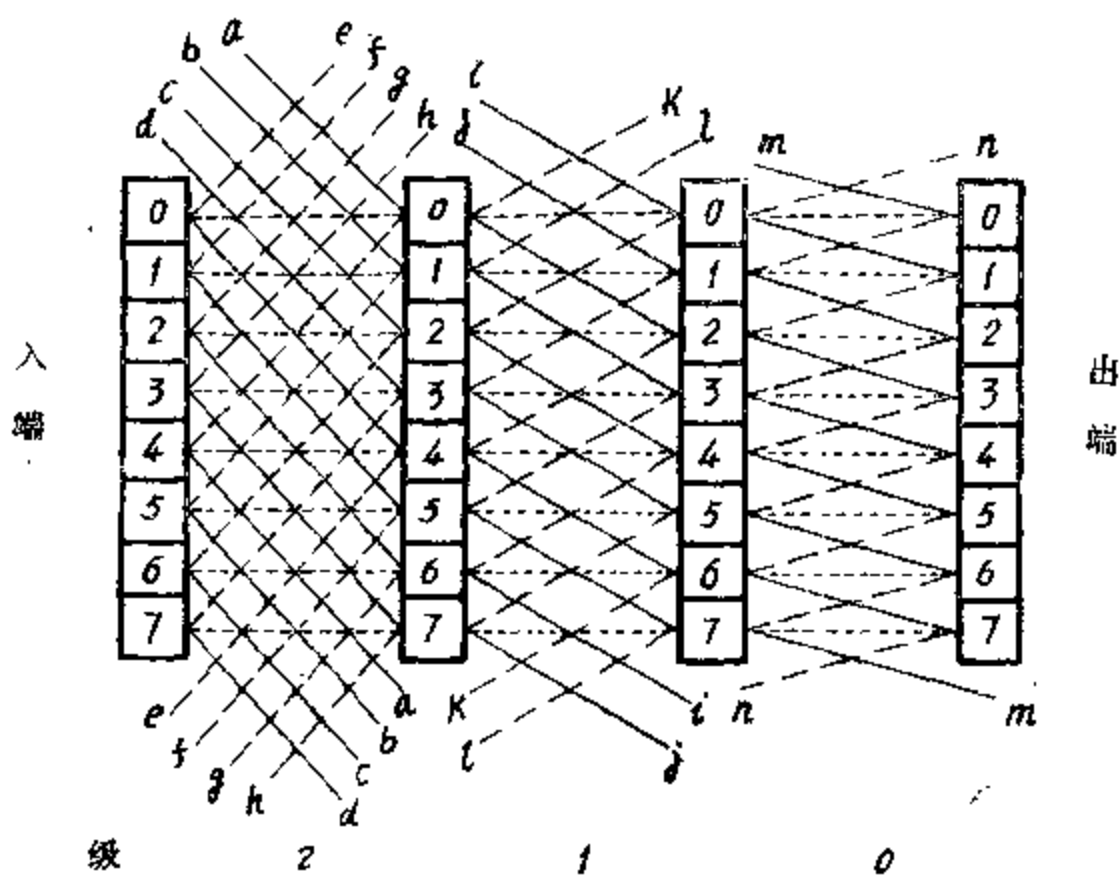


图 7.25 $N=8$ 多级 PM2I 网络

ADM 的拓扑和控制结构使它可以完全模仿 ω 网络的四功能交换单元。拿 x, y ($x < y$) 两个入单元来说, H_x, H_y 为直连, D_x, U_y 为交换, U_y, H_y 为下播, D_x, H_x 为上播。因此, ADM 可以实现 ω 网络的全部连接, 而且其组合数还要更多。利用数据变换网络实现各种灵活的移数、重复、间隔、展开等变换函数的例子见文献[23]。

比较上述四种多级网络, 灵活性由低到高的次序是: STARAN, 间接二进制 n 方体、 ω 、ADM, 而复杂性和成本的次序则与此相反。虽然这些网络的设计者都提出了各自的网络用途 (例如 STARAN 网络和 ω 网络都是为了进行存储器与处理器之间的数据变换, 间接二进制 n 方体网络是为了连接成微处理器阵列), 但从上面对各种网络共同性的分析可以看出, 它们对多种应用场合都是适合的。

§ 3 多处理机

本节首先分析多处理机区别于并行处理机的显著特点, 从而把关于多处理机的介绍限制在这几个方面的问题上, 即结构灵活性、程序并行性、并行任务派生与同步、资源分配和任务调度等。与并行处理机的叙述不同, 我们将不具体介绍某一台多处理机的详细结构, 而只说明一般原理, 这是从多处理机目前发展还不很成熟这一情况出发的。

3.1 多处理机与并行处理机的区别

多处理机属于多指令流多数据流系统，它和上一节叙述的属于单指令流多数据流系统的并行处理机相比，有很大的差别，主要有以下几方面：

(1) 结构灵活性

并行处理机带有专用性，主要针对特定的算法设计其结构，其特点是处理单元数量很多，但只需设置较为有限和固定的机间互连通路，即可满足一批并行性很高的算法的需要。而多处理机应有较强的通用性，也不止于向量数组处理。因此，为适应多样的算法，应创造更为灵活的系统结构，以实现各种复杂的机间互连模式。目前，多处理机中处理机的数目还不可能做得很多，与并行处理机中处理单元数已达到 16384 相比，还差三个数量级。

(2) 程序并行性

同样由于专用性的原因，加上并行处理机的并行性存在于指令内部，所以并行性的识别是比较容易的，主要在指令类型及硬件结构上即已考虑，可由程序员在编制程序中加以掌握。但在多处理机中，并行性存在于指令外部，即表现在多个任务之间。为充分发挥系统通用性的优点，便要利用多种途径——算法、程序语言、编译、操作系统以至指令、硬件等——尽量挖掘各种潜在的并行性，而且主要的责任不能放在程序员肩上。

(3) 并行任务派生

并行处理机依靠单指令流对多数据组实现并行操作，这种并行操作是通过各条单独的指令加以反映和控制的。但多处理机是处于多指令流操作方式，一个程序当中就存在多个并发的程序段，需要专门的指令来表示它们的并发关系以及控制它们的并发执行，使一个任务正在被执行时就能派生出可与它并行执行的另一些任务，这称为并行任务派生(Parallel task Spawning)。

(4) 进程同步

并行处理机实现操作级的并行，所有处于活动状态的处理单元同时执行同一条指令操作，受同一个控制器控制，工作自然是同步的。但多处理机实现指令、任务、程序级的并行，一般说，在同一时刻，不同的处理机执行着不同的指令，由于执行时间互不相等，它们的工作进度不会也不必保持相同。但是，如果并发程序之间有数据交往或控制依赖，就必须采取特殊的同步措施，使它们包含的指令相互间仍保持程序要求的正确顺序。

(5) 资源分配和任务调度

并行处理机主要执行向量数组运算，处理单元数目是固定的，程序员据以编写程序，并只能利用屏蔽手段设置处理单元的活动状态来改变实际参加并行操作的处理单元数目。但多处理机执行并发任务，需用处理机的数目没有固定要求，各个处理机进入或退出任务以及所需资源变化的情况都要复杂得多，这就提出了一个资源分配和任务调度问题，解决的好坏对整个系统的效率有很大的直接影响。

我们在下面分别对这些问题进行讨论。

3.2 多处理机硬件系统结构

与上一节讨论并行处理机一样，机间互连机构仍然是决定多处理机系统性能的重要因素之一，但二者的着眼点又有所区别。上一节介绍的机间互连网络或者是针对处理单元数量很

大, 连接方式较受限制的情况, 或者是要达到数据变换的目的。多处理机的机间互连, 除了高频带、低成本等共同要求外, 还要满足另外的特殊要求: 一是机间通信模式的多样性, 要求实现方式更为灵活的连接; 二是机间通信的不规则性, 要求实现无冲突的连接。这些在处理机数量很大的情况下都是难以得到满意解决的课题。因此, 到目前为止实现的多处理机大多是规模较小的系统, 采用的机间互连结构形式与上一节的 SIMD 互连网络是不同的。下面将以较粗的划分讨论几种结构形式:

- 总线结构;
- 交叉开关结构;
- 多端口存储器结构;
- 开关枢纽结构。

虽然具体结构形式更为多样, 复杂性也各异, 但从系统核心部分的基本特征来说, 大都可以归纳到这几类中去。

3.2-1 总线结构

总线结构是多处理机最简单的一种结构形式, 就是多台处理机 P 包括它们的内存 M 和外围设备 I/O 通过自身的接口用一套总线相连, 构成所谓分时总线或公共总线系统, 如图 7.26 所示。其中, (a) 是利用双向总线; (b) 是利用单向总线。后者由控制部件 (CU) 进行集中控制。二者都是单总线系统。

总线系统只利用高频电缆或双绞导线作为总线, 而不增加其它开关、放大器等有源元件, 所以成本低, 工作可靠。机间通信是通过发送和接收双方的总线接口来进行的, 例如, 发送端首先测试总线状态, 如果空闲, 就可送出目的处理机地址, 测试它的状态, 在它已准备好接收的条件下, 把数码和有关控制信息传送过去。接收端只需鉴别信息的源处理机地址, 并对控制信息作出必要的应答。

总线结构另外的优点是便于利用工业定型生产的现成硬件, 且处理机数目的增减和设备类型的改变

在总线频带允许的范围内比较容易实现。但是, 随之而来的问题是总线竞争, 使机间信息传输频带和反应速度都受到很大的限制。在单总线结构中, 由于同一时间内只允许一对处理机相互通信, 处理机数目增加, 自然不得不对整个系统的性能产生严重影响。

实际的多处理机如果采用总线结构原理, 都要寻求各种新的改进方案。其中最基本的是采用多总线。具体方案可以是重复设置多套总线, 增加处理机之间可能利用的通信路径, 从而成倍地加宽其有效传输频带; 也可以是按性质分别设置处理器总线、存储器总线、 I/O 总

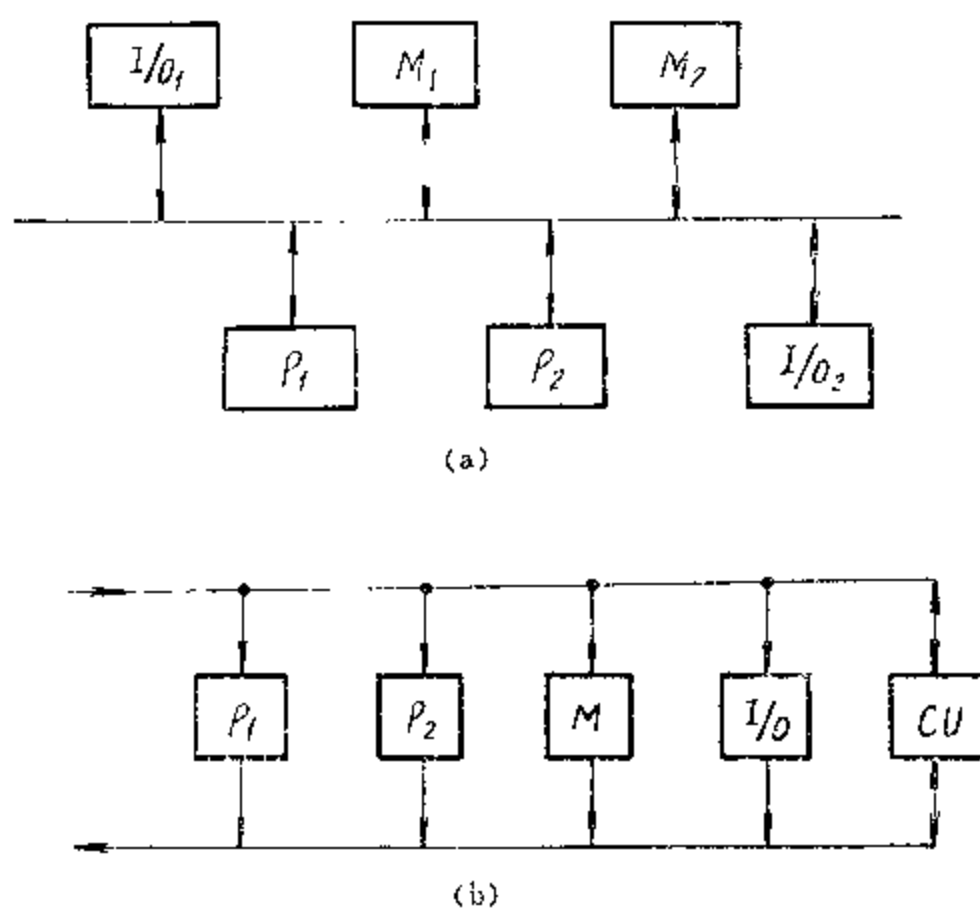


图 7.26 双向(a)和单向(b)总线系统

线等，再利用总线耦合器把它们连在一起。

尽管总线结构在扩大系统规模上似乎有较大的局限性，但在新近发展的不少多处理机系统中，它仍是获得广泛采用的一种主要结构形式。究其原因，第一是总线结构本身的改进，第二是采用微处理机发展分布处理系统的影响。下面举几个例子。

日本的实验多处理机系统 EPOS 采用了典型的多总线结构，如图 7.27 所示。由四个系统总线控制器 SBC 所控制的四套总线使允许连接的处理机 P 和 I/O 通道控制器 (I/O C) 的总数可达 64。这些处理机和 I/O 通道控制器都是模块化的，尽可能利用自己的本地内存，也可选择一套总线去访问其它模块的本地内存或共享公共内存 M_0 、 M_1 ，因此这符合分布处理系统的原则。系统中采用 7000 门的大规模集成电路处理器片 PULCE，这表明 EPOS 是一个反映现代水平的多处理机系统。

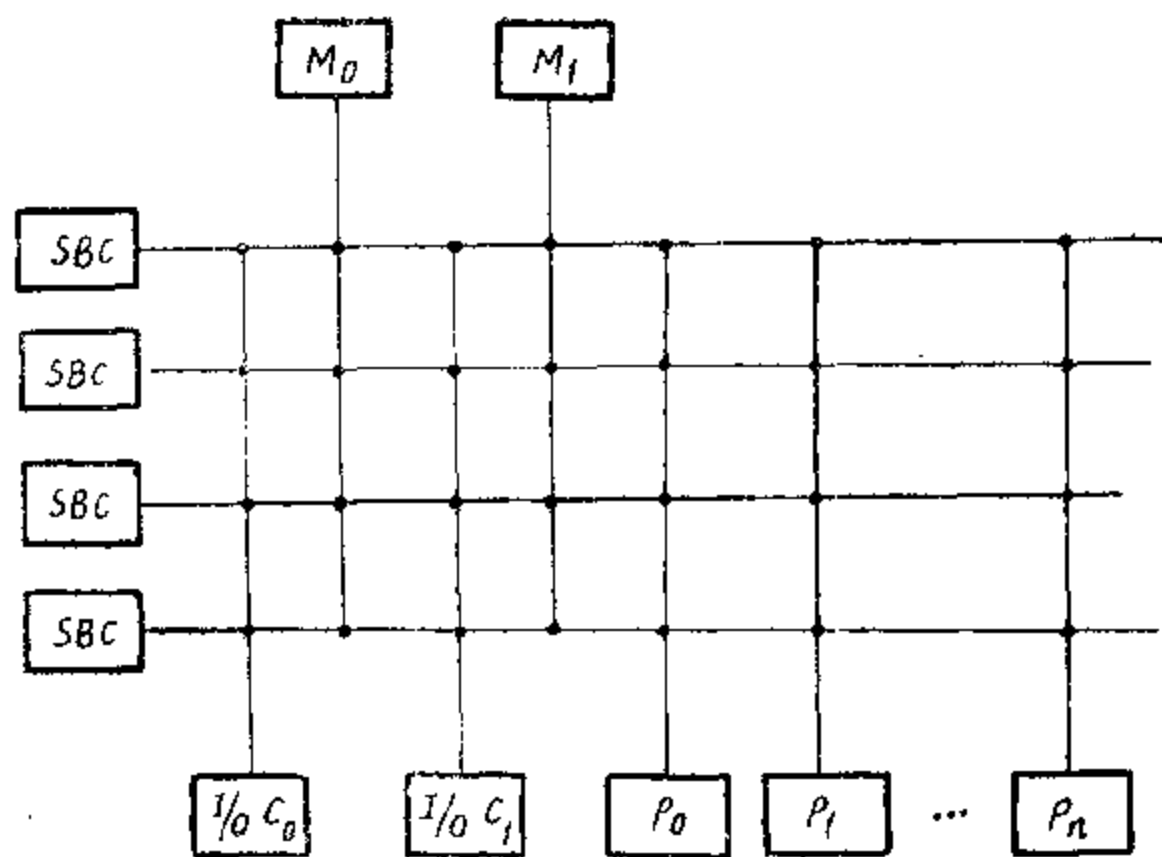


图 7.27 EPOS 多处理机结构框图

西德西门子公司研制的结构式多处理机系统 SMS 采用分级的多总线结构，如图 7.28 所示。由一台高档小型计算机充当的主机通过接口和八个总线驱动器驱动 8 套总线，每套总线上连接 16 台微处理机，组成一个分布式多微处理机系统。1977 年开始运行的 SMS201 机利用 128 台普通的 8 位微处理机实现了一台中型单处理机的运算速度，即每秒 32×10^6 次 8 位运算，折合 64 位运算每秒一百万次以上。SMS 201 对浮点运算的速度是有限的，即使增加快速算术部件，也只能做到每秒 50 万个 32 位浮点结果。为了提高这项指标到 1800 万，新设计的 SMS 3 多处理机采取了一系列措施，包括利用位片式微处理机、增加字长和采用双总线结构等。

美国的 C_m^* 多微处理机²⁴ 采用机群结构，如图 7.29 所示。虽然它有许多具体的特点，但从结构形式上，仍可视为分级多总线结构一类。已经装成的系统共包含 50 台 DEC LSI-11 微计算机，用三级总线连接起来。最基层的一级是计算机模块 CM 本身的 LSI-11 总线，但是内部插入了一个本地开关 S，通过它使处理器 P 或者与自己的本地存储器 M 联系，或者通向第二级总线。这个第二级总线称为 Map 总线，每一机群一套，它把机群内部

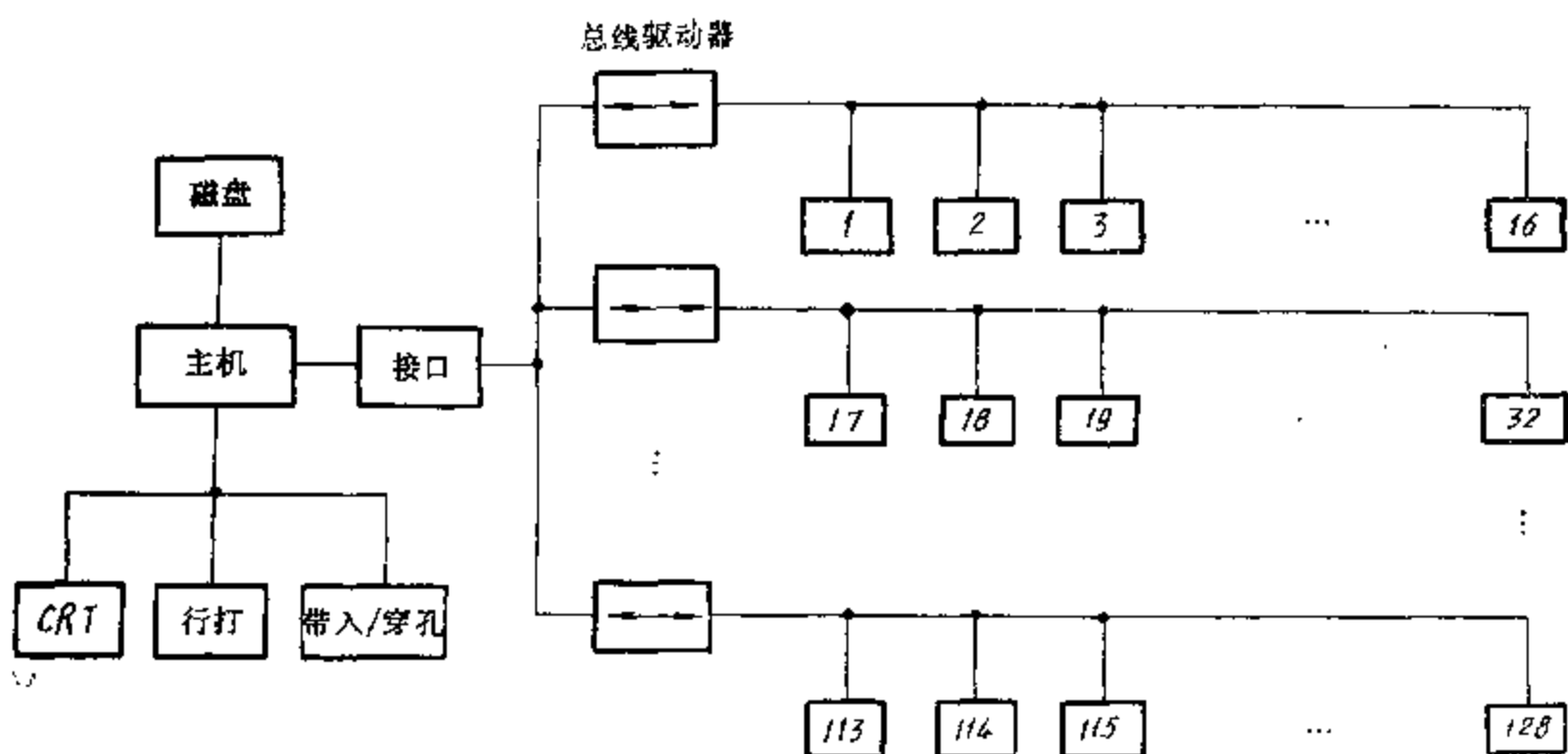


图 7.28 SMS 多处理机结构框图

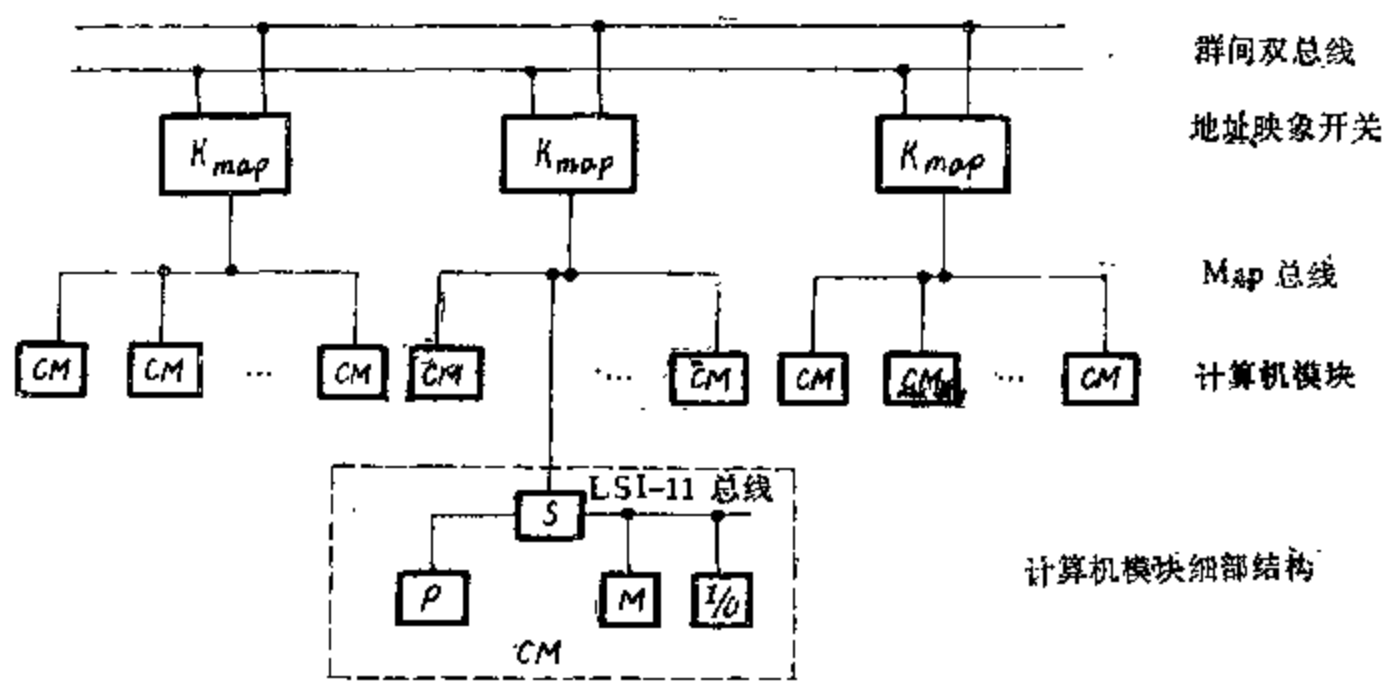


图 7.29 C_m^* 多处理机结构框图

的计算机模块 CM（最多可达 14 个）连接起来，通向地址映象开关 K_{map} 。由于 C_m^* 是一个新型的分布处理系统，各个计算机模块的 28K 字本地内存组织在一起，形成统一的具有 28 位地址的虚拟存贮空间，地址映象开关就担负主要的地址变换任务。不同机群之间再通过各自的 K_{map} 利用第三级总线相连。为了加宽频带，这个第三级总线采用了双总线结构。本地、机群内和机群间三级的存贮器访问时间分别是 $3.5\mu s$ ， $9.3\mu s$ 和 $26\mu s$ 。虽然它们以大约 3 倍的比值变慢，但由于程序的局部性原则，使本地访问的命中率能达到 90% 左右，从而使平均的存贮器访问时间缩短为 $4.1\mu s$ ，显示了分布处理系统的优越性。

3.2-2 交叉开关结构

交叉开关结构包含一组纵横开关阵列，把横向的处理器 P 及 I/O 通道与纵向的存贮器模块 M 连接起来，如图 7.30 所示。

交叉开关结构可以看作是多总线结构朝总线数量增加的方向发展的极端情况。在这种情

况下, 总线套数 = $n + i + m$, 且 $n \geq i + m$, 使在同一时间 m 个处理器加上 i 个 I/O 通道中每一个都能分到一套总线, 与 n 个存储器模块中的任一个相连。因此, 交叉开关大大展宽了传输频带和提高了系统效率。但是, 图 7.30 中每一个交叉点实际都代表一套开关, 而且包含必要的用来分解多重访问冲突的硬件设备; 如果再考虑到总线的宽度, 整个交叉开关阵列的复杂程度是可想而知的。通常估算时, 认为交叉开关阵列的设备量是 $O(n^2)$, 当 n 很大时, 其成本可能会超过全部 $2n$ 台处理器、存储器和 I/O 设备的成本。从目前实际做成的系统来看, 比较合理的规模是 $n \leq 16$, 个别的有达到 $n = 32$ 的。例如, 美国的 C.mmp 多处理机和 S-1 多处理机都是各包含 16 台处理机采用交叉开关结构的系统。

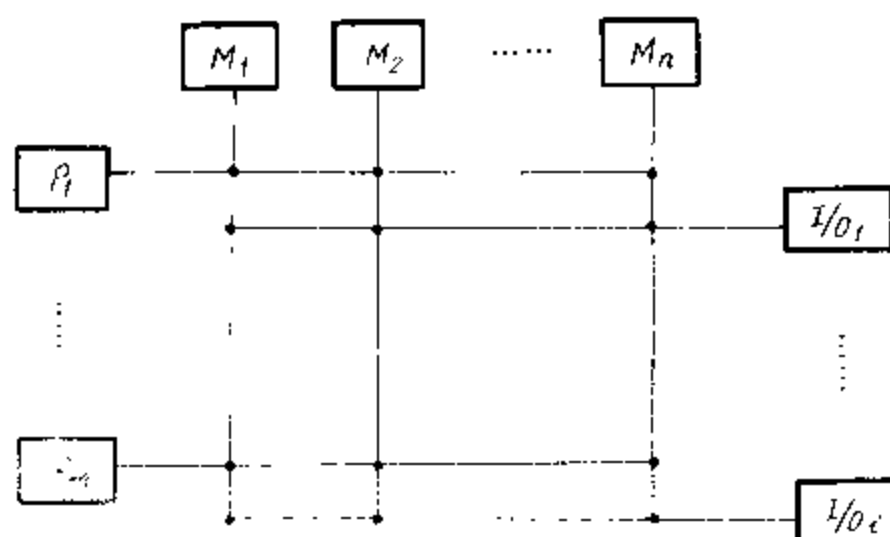


图 7.30 交叉开关结构

3.2-3 多端口存储器结构

多端口存储器结构的中心是多端口存储器模块 M 。多个模块的相应端口连在一起, 每一个端口负责处理一个处理器 P 或 I/O 通道的存储器访问请求。一个四端口存储器系统结构如图 7.31 所示。对照图 7.30 可以看出, 如果把交叉开关结构中分布在各个交叉点上的开关和控制逻辑都移到相应存储器模块的接口内部来, 那么就得到图 7.31 的结构形式。每一存储器模块按照对它的各个端口指定的优先级来分解对它的存储器访问冲突。与交叉开关结构的全部复杂性集中在开关阵列一样, 现在, 在多端口存储器结构中, 全部系统的复杂性就移到了存储器模块中来。因此, 它的端口数不宜做得太多, 而且一经做定, 就不易改变, 这都是对组成多处理机不利的因素。但是, 当处理机数目不多时, 多端口存储器结构还是用得比较成功的。例如, 最早的工业定型多处理机 UNIVAC-1108 就是一个这样的系统, 用五端口存储器组成。现在 UNIVAC-1100 系列已经发展到 1100/80, 它是一台包含两重并行性的多处理机系统。从外部看, 系统由多至 4 台对称的多处理机组成; 而从内部看, 它又是在微程序一级用多微处理机实现流水线结构的并行系统。

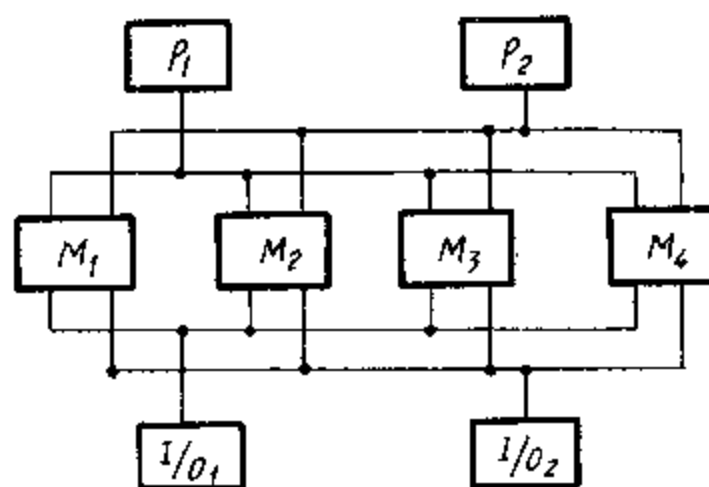


图 7.31 四端口存储器系统结构

3.2-4 开关枢纽结构

无论是总线, 或是交叉开关, 或是上一节所讨论的 SIMD 互连网络, 都有一个共同点, 就是有多个入端和多个出端, 要在它们之间进行转接, 使入端有选择地和出端相连。我们把这种功能部件通称为开关枢纽 (tie)。既然有多个入端, 就必然存在互连要求上的冲

突,因此开关枢纽要包含相应的分解冲突的部件,这称为仲裁单元。仲裁单元与在一个入端和多个出端间进行转接的开关单元一起,就构成一个基本的开关枢纽。一般而言,任何互连网络都是由一个或多个开关枢纽组成的。利用这一普遍的观点,可以把总线、交叉开关和SIMD互连网络(以二级立方体网络为例)的结构表示为图7.32的开关枢纽形式,其中a表示仲裁单元,是多个入端到一个出端,而s表示开关单元,是一个入端到多个出端。由图7.32可见,总线的a单元和s单元之间只有一条通路,说明它的频带较窄;而交叉开关频带虽宽,但a单元和s单元的扇入和扇出值都很大,说明设备较复杂;至于SIMD互连网络的画法,是用s-a结构表示一个交换单元,它和一个 2×2 的交叉开关是相同的。

利用交叉枢纽作为基本构件,可以把每台处理机与周围邻近的处理机连接起来,组成各种具有分布结构的多处理机。开关枢纽的选择应使组成的多处理机具有较佳的拓扑结构和良好的互连特性,特别是要适应处理机数量很多的情况。

理想的拓扑结构应该是每个开关枢纽的端数不宜太多,但所用数量要少,能以较短的路径把数量很大的处理机连接起来,实现快速灵活的相互通信。美国在设计中的树形多处理机X-TREE^[25]就是一个很好的例子。它的结构框图如图7.33所示。N个处理机(称为X-结点)被连接成 $\lceil \log_2 N \rceil$ 级的二叉树,使结点间的平均路径长度与结点数目的对数成正比。N越大时,越显示出

它的优越性。如果在二叉树的每一级内增加水平连接线,则构成半环二叉树(图7.33实线)或余环二叉树(图7.33实线加虚线)。这样,不但可进一步缩短通信路程,平衡各结点上的信息流量,而且还能提高路径的冗余度以达到容错的目的。半环二叉树要求X-结点内部包含一个五端的开关枢纽S进行本结点与其外部四个邻近结点间的连接。而对树叶结点来说,则与共享存储器模块 M_s 和I/O装置相连,如图7.33所示。所有X-结点的硬件都做在一块超大规模集成电路片上。对于内存容量为16K至128K字节的X-结点,需要达到的集成电路元件密度是每片292000至1622000个门。

多处理机的结构型式变化很多,它要满足一系列要求,才能使系统具有通用性的特点,其中主要的有:模块性、可重构性、可分块性、紧密耦合、多工作模式等。在处理机数目很多的系统中,解决通用性问题的可能途径是将SIMD模式与MIMD模式结合起来,多任务与多道程序结合起来,而且采用模块性较好的分布结构^[26]。

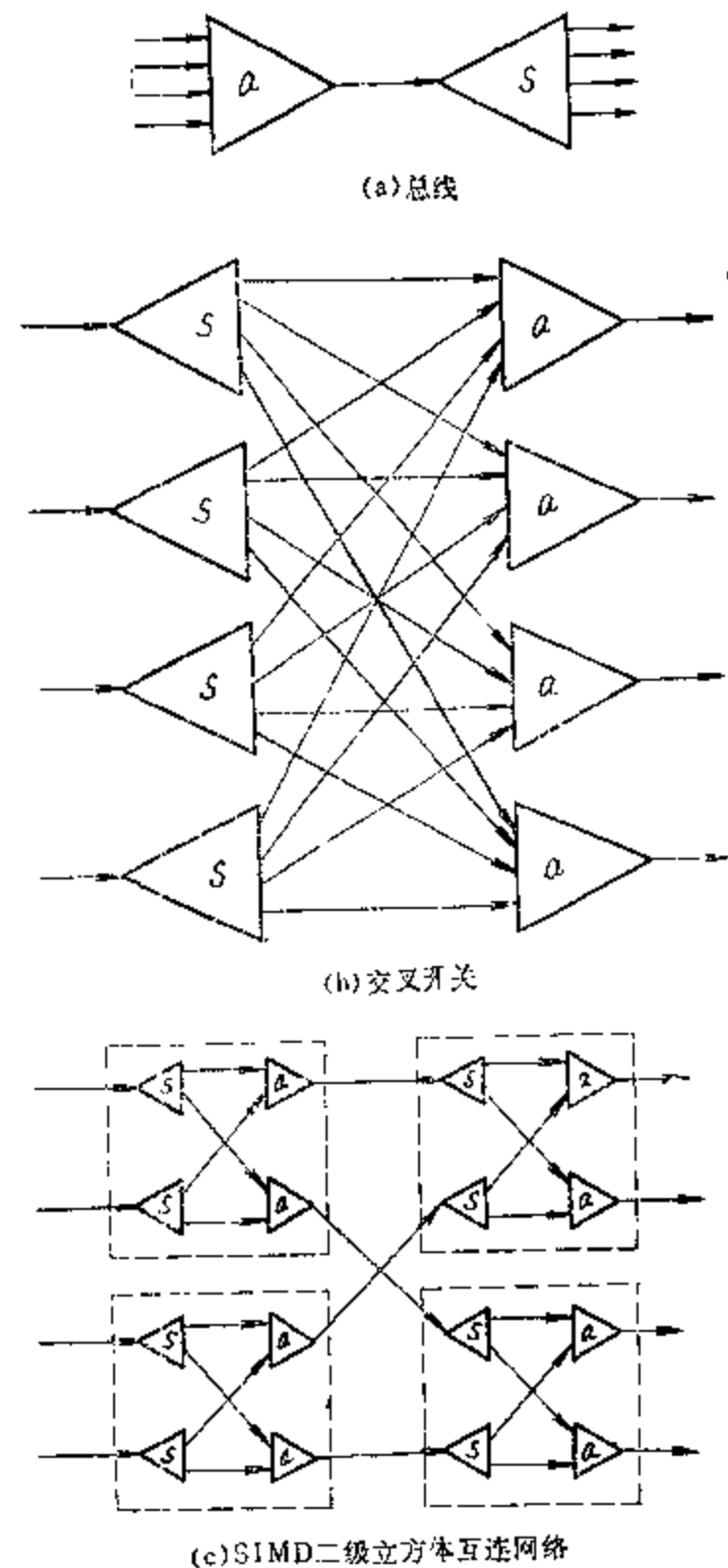


图 7.32 总线、交叉开关和 SIMD 互连网络

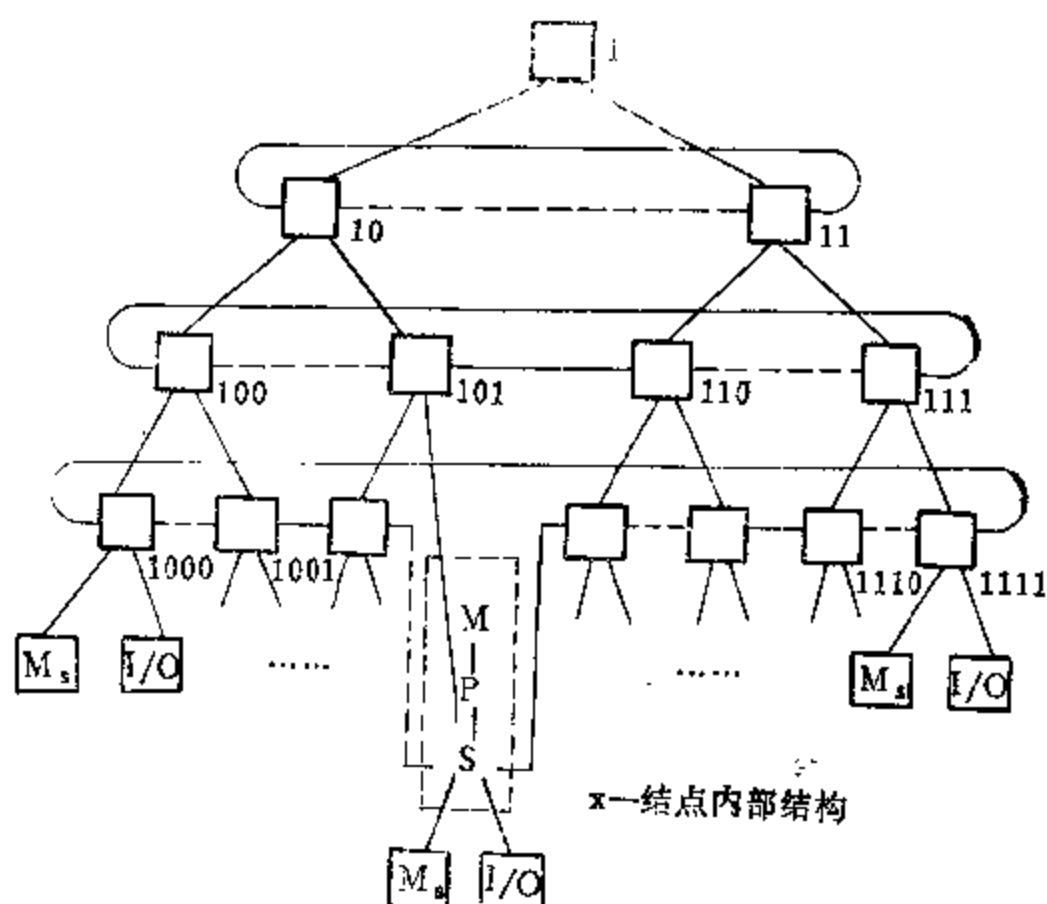


图 7.33 X-TREE 多处理机结构框图

3.3 程序并行性

并行性的关键在算法。特别是在处理机数目很多时，要把任何一个问题分解成足够多的并行过程是极为困难的。除了一些已为 SIMD 处理机所解决了的问题（主要是向量数组运算）以外，并不是所有问题都能做到这一点。这方面目前虽有一些研究成果，但和多处理机本身一样，还没达到普遍应用的程度。

3.3-1 算术表达式的并行运算

有许多例子可以说明，算法必须适应具体的计算机结构。串行处理机上习惯采用的循环、迭代解法，往往不适用于多处理机，而采用直接解法，有时反而能揭示更多的并行性。试比较同一多项式的下列两种表达式：

$$E_1 = a + bx + cx^2 + dx^3 \quad (7.3-1)$$

$$E_2 = a + x\{b + x[c + x(d)]\} \quad (7.3-2)$$

后一式是根据霍纳(Horner)法则得到的串行处理机上的典型算法，共需 3 个乘加循环，6 级运算；但是，它对多处理机并不适合，使任何一台增加的处理机毫无用武之地。倒是采用前一式的解法更加有效，只需 4 级运算就够了。这二式的运算过程表示为树形流程图，分别如图 7.34(a)和 (b) 所示。运算的级数就称为树高，用 T_p 代表， P 为所需处理机的数目。 T_p 与顺序运算级数 T_1 的比值就称为加速比，用 S_p 代表；而 $S_p/P = E_p$ 就称为效率。可见， $S_p \geq 1$ ，而 $E_p \leq 1$ ，即运算的加速总是伴随着效率的降低。研究如何将算术表达式变形，使树高减少，这是并行算法研究的重要课题之一^[1]。

首先从算术表达式的最直接形式出发，利用交换律把相同的运算集中在一起，再利用结合律把参加这些运算的操作数（称原子）配对，尽可能并行运算，从而组成树高最小的子树，最后再把这些子树结合起来。例如给定表达式

$$E_2 = a + b(c + def + g) + h \quad (7.3-3)$$

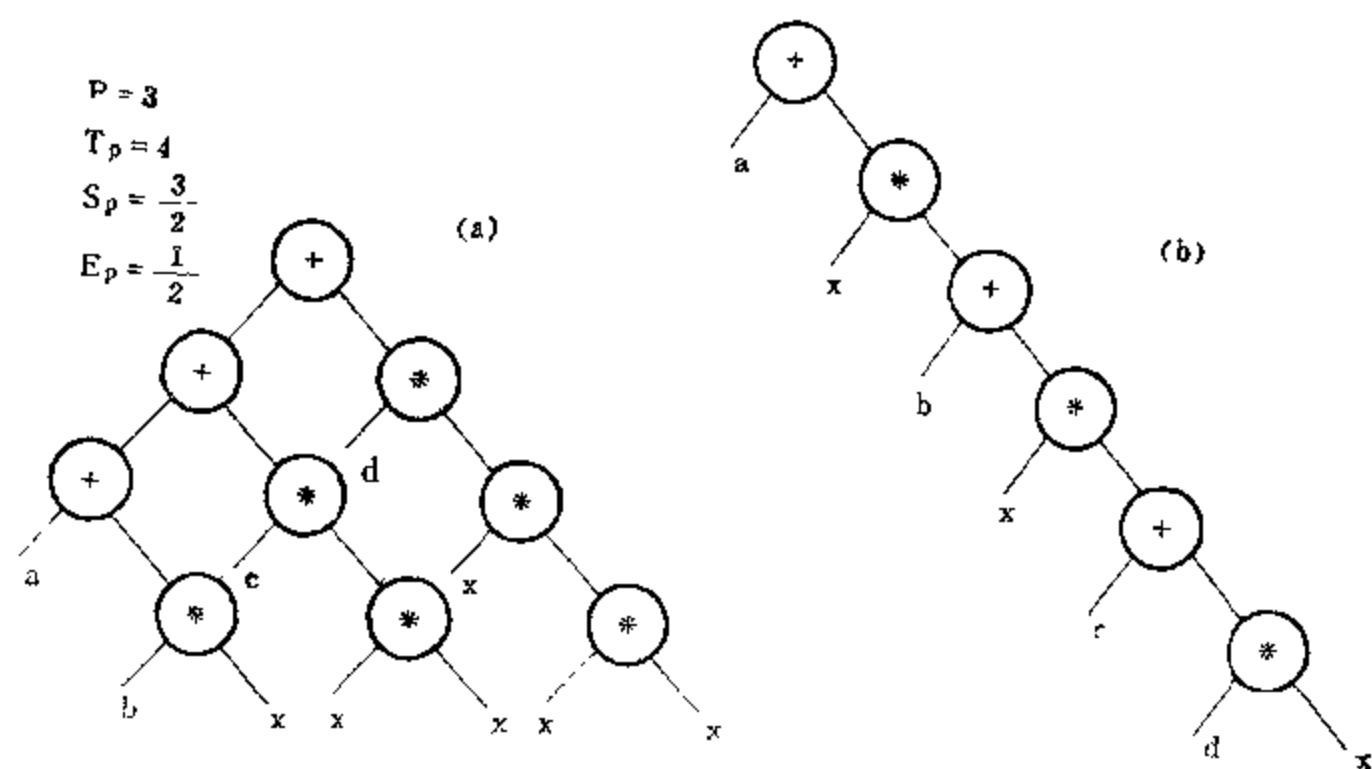


图 7.34 不同算法影响树高的例子

需 7 级运算，如图 7.35(a) 所示。利用交换律和结合律改写为

$$E_2 = (a + h) + b[(c + g) + def] \quad (7.3-4)$$

则只需 5 级运算，如图 7.35(b) 所示。

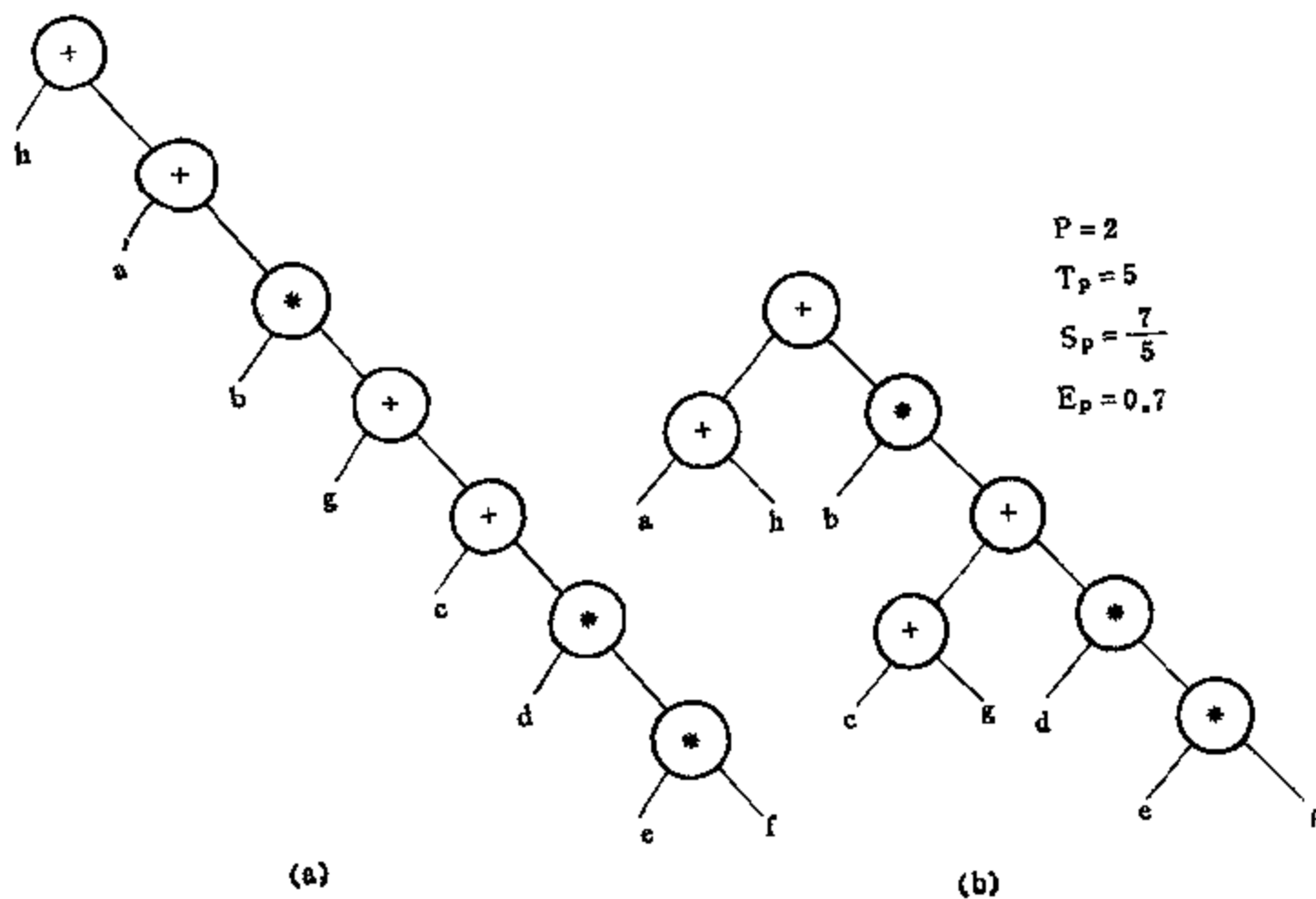


图 7.35 利用交换律和结合律减少树高

进一步减少树高，需利用分配律，在恰当平衡各子树的级数的情况下，往往能收到较好的效果。例如上式计算 $(c + g)$ 的子树只用一级，而计算 def 的子树要用 2 级，相加乘 b 需再增加 2 级；如果把 b 写进括号内，则计算 $bdef$ 仍用 2 级已够，却省去了后来的一次乘 b ，使总级数由 5 减为 4。将式 (7.3-3) 改写如下：

$$E_2 = (a + b) + (bc + bg) + bdef \quad (7.3-5)$$

运算过程如图 7.36 所示。

运算表达式并行性的识别，除了依靠算法以外，还可以依靠编译程序。有一些编译算法^[8]可以直接从给定的算术表达式产生能并行执行的机器指令，经过或不经过波兰后缀表达式。例如，给定算术表达式

$$Z = E + A * B * C / D + F \quad (7.3-6)$$

利用普通的串行编译算法，产生出三元指令组为：

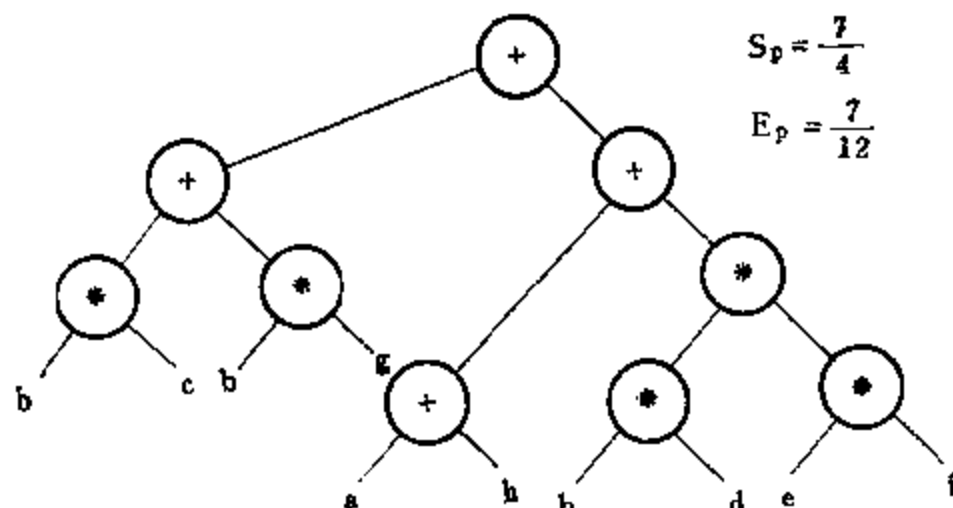


图 7.36 利用交换律、结合律和分配律减少树高

1	*	A	B
2	*	1	C
3	/	2	D
4	+	3	E
5	+	4	F
6	=	5	Z

指令之间都是相关的，需 5 级运算。如采用并行编译算法，则可得到能并行执行的指令组：

1	*	A	B
2	/	C	D
3	*	1	2
4	+	E	F
5	+	3	4
6	=	5	Z

分配给两个处理机，只需三级运算。

可见，有了好的并行编译算法，算术表达式的预先变形也可以是不必要的。

3.3-2 递归程序的并行性

除了算术表达式以外，递归程序的并行性是研究的另一重要课题，但是它更要困难一些。这是因为前者属于静态标量问题，后者属于动态标量或向量问题，就好象前者对应组合逻辑问题，后者对应时序逻辑问题一样。

关于程序递归关系的并行性，将只讨论线性递归，它代表了实际中遇到的大部分情况。下面是一些线性递归的例子：

(1) 给定向量

$$a = (a_1, a_2, \dots, a_n)$$

$$b = (b_1, b_2, \dots, b_n)$$

求欧几里得内积

$$a \cdot b = a_1 b_1 + a_2 b_2 + \dots + a_n b_n \quad (7.3-7)$$

这归结为下列递归关系

$$\left. \begin{aligned} x_0 &= 0 \\ x_i &= x_{i-1} + a_i b_i \quad 1 \leq i \leq n \end{aligned} \right\} \quad (7.3-8)$$

(2) 计算多项式

$$p_n = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n \quad (7.3-9)$$

写成霍纳法则形式, 归结为下列递归关系

$$\left. \begin{aligned} p_0 &= a_0 \\ p_i &= a_i + x p_{i-1} \quad 1 \leq i \leq n \end{aligned} \right\} \quad (7.3-10)$$

(3) 计算菲波纳西 (Fibonacci) 数列, 可利用下列递归关系:

$$\left. \begin{aligned} f_1 &= f_2 = 1 \\ f_i &= f_{i-1} + f_{i-2} \quad 3 \leq i \leq n \end{aligned} \right\} \quad (7.3-11)$$

(4) 计算 n 位二进制数 $a = a_n \cdots a_1$ 和 $b = b_n \cdots b_1$ 相加时的进位, 可利用下列递归关系:

$$\left. \begin{aligned} c_0 &= 0 \\ c_i &= a_i \wedge b_i \vee (a_i \vee b_i) \wedge c_{i-1} \quad 1 \leq i \leq n \end{aligned} \right\} \quad (7.3-12)$$

以上这些线性递归的例子, 可以归纳为下列线性递归普遍式:

$$\left. \begin{aligned} x_i &= 0 \quad i \leq 0 \\ x_i &= c_i + \sum_{j=i-n+1}^{i-1} a_{ij} x_j \quad 1 \leq i \leq n \end{aligned} \right\} \quad (7.3-13)$$

用矩阵形式表示,

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{bmatrix} + \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ a_{21} & 0 & 0 & \cdots & 0 \\ a_{31} & a_{32} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \quad (7.3-14)$$

$$\text{或} \quad X = C + AX \quad (7.3-15)$$

式中, $C = (c_1 \cdots c_n)^t$ 称为常数向量; A 是一个严格下三角矩阵, 称为系数矩阵; $X = (x_1 \cdots x_n)^t$, 称为结果向量; t 表示向量转置。

在多台处理机上求解这个线性递归普遍式, 可以采取下面的方法。

第一种方法称为列扫描法, 步骤如下:

第一步先算 $x_1 = c_1$, 并将 x_1 播送到其余各式, 用 $(n-1)$ 个处理机计算 $a_{21}x_1, a_{31}x_1, \cdots, a_{n1}x_1$;

第二步用 $(n-1)$ 个处理机计算 $x_2 = c_2 + a_{21}x_1, c_3 + a_{31}x_1, \cdots, c_n + a_{n1}x_1$;

第三步把 x_2 播送到其余各式, 用 $(n-2)$ 个处理机计算 $a_{32}x_2, a_{42}x_2, \cdots, a_{n2}x_2$;

第四步用 $(n-2)$ 个处理机计算 $x_3 = c_3 + a_{31}x_1 + a_{32}x_2, \cdots, c_n + a_{n1}x_1 + a_{n2}x_2$;

.....

依此类推, 直到计算出全部 n 个结果。

由上可见, 这个方法需用 $(n-1)$ 个处理机计算 $2(n-1)$ 步。如果 $n=4$, 则需 3 个处

理机用 6 步运算完成之。

第二种较快的并行算法称为乘积形式递归算法，它是根据严格下三角系数矩阵 A 的性质把线性递归普遍式 $X = C + AX$ 写成下列形式：

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} c_1 \\ (c_2 + a_{21}c_1) \\ (c_3 + a_{31}c_1) + a_{32}(c_2 + a_{21}c_1) \\ (c_4 + a_{41}c_1) + a_{43}(c_3 + a_{31}c_1) + (a_{42} + a_{43}a_{32})(c_2 + a_{21}c_1) \\ \dots\dots\dots \end{bmatrix}$$

例如，当 $n=4$ 时，式中右边部分括号内的值只有 4 种是不同的，需用 4 个处理机经 2 步同时算出，再用 2 个处理机经 3 步算出 x_3 、 x_4 的最后结果。总计起来，比上一算法少用了一步的时间，但多用一个处理机。当 n 较大时，这个算法与上算法相比，快速性会更为明显，但处理机的数目也会增加较多。

实际中应用这些方法，需先分析程序的递归结构，化成线性递归普遍形式。举例如下，给定程序段

```

DO 4 I=1, N
1   E(I)=3*F(I)+SIN(P(I))
2   B(I)=D(I-1)+Q(I)
3   D(I)=E(I)+B(I)
4   CONTINUE

```

这个程序段的数据相关图示于图 7.37 中。实际上，语句 1 只是为语句 3 提供数据 $E(I)$ ，而且可在进入循环前全部算出，故可认为系属一个单独的数组运算。真正构成闭合循环的只是语句 2 和语句 3，展开如下：

```

2   B(1)=D(0)+Q(1)
3   D(1)=B(1)+E(1)
2   B(2)=D(1)+Q(2)
3   D(2)=B(2)+E(2)
2   B(3)=D(2)+Q(3)
3   D(3)=B(3)+E(3)
.....

```

(7.3-16)

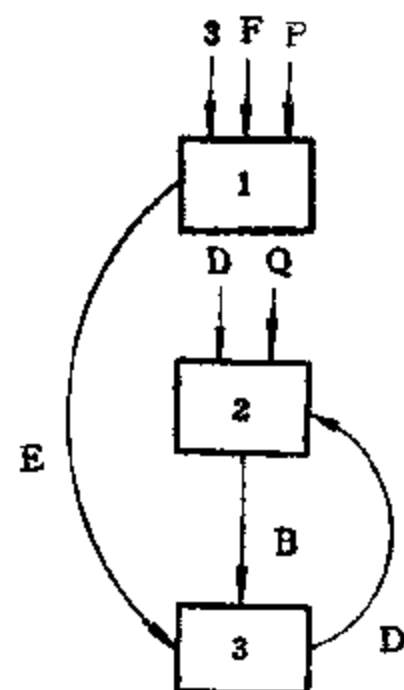


图 7.37 线性递归程序数据相关图的例子

这样，就把原来的循环程序变成了包含 $2N$ 个语句的线性递归普遍形式：

$$\left. \begin{aligned} x_1 &= x_0 + c_1 \\ x_2 &= x_1 + c_2 \\ x_3 &= x_2 + c_3 \\ x_4 &= x_3 + c_4 \\ x_5 &= x_4 + c_5 \\ x_6 &= x_5 + c_6 \\ &\dots\dots\dots \end{aligned} \right\} \quad (7.3-17)$$

式中

$$\begin{aligned}x_0 &= D(0) \\x_{2i-1} &= B(i) \\x_{2i} &= D(i) \\c_{2i-1} &= Q(i) \\c_{2i} &= E(i) \\1 \leq i \leq N\end{aligned}$$

如果 $SIN(P(I))$ 也算作一步运算, 则语句 1 需 N 个处理机 3 步运算; 语句 2 和 3 的循环虽共包含 $2N$ 个语句, 但由于是累加和的形式, 用 N 个处理机也就够了。

3.3-3 程序并行性的分析

几个任务能否并行执行, 不但取决于算法, 还与程序的结构形式有很大的关系。通常为了加强程序并行性的识别能力, 有必要在程序语言中增加能明确表示并发进程的成分。在标准语言的基础上加以扩充, 或是重新设计专门的并行语言, 这是当前采取的两种不同途径。实际的多处理机中总是两类语言并存, 这是与现代计算机保持尽可能多种可用语言的做法一致的。

下面只对程序并行性作一般概念上的介绍^[1, 28]。假定有一个程序包含 $P_1, P_2, \dots, P_i, \dots, P_j, \dots, P_n$ 等多个程序段, 按照书写的格式当然是串行的, 表示了它们的正常执行顺序。为了分析的方便, 我们取最简单的情形, 即 P_i 和 P_j 都是一条语句, P_i 在 P_j 之前执行; 且只讨论 P_i 和 P_j 的直接数据相关关系。实际上, P_i 和 P_j 即使表面上无关, 但可能通过它们之间的其它语句形成间接数据相关关系, 也就是说, 下面讨论的原理在实际应用时应适当推广。一般说来, P_i 和 P_j 之间存在三种可能的数据相关情况:

(1) 如果 P_i 的左部变量也在 P_j 的右部变量集内, 且 P_j 要从 P_i 取得算出的值, 则称 P_j “数据相关”于 P_i 。例如,

$$\begin{aligned}P_1 \quad & A = B + D \\P_2 \quad & C = A * E\end{aligned}$$

P_2 必须取 P_1 算得的 A 值作为操作数。

(2) 如果 P_j 的左部变量也在 P_i 的右部变量集内, 则称 P_i “数据反相关”于 P_j 。例如,

$$\begin{aligned}P_1 \quad & C = A * E \\P_2 \quad & A = B + D\end{aligned}$$

在 P_1 未取用 A 量之前, A 的值不能被 P_2 所改变。

(3) 如果 P_i 的左部变量也是 P_j 的左部变量, 则称 P_j “数据输出相关”于 P_i 。例如,

$$\begin{aligned}P_1 \quad & A = B + D \\P_2 \quad & A = A + C\end{aligned}$$

P_2 存入它自己算得的值必须在 P_1 存入之后。

除这三种情形之外, 便是 P_i 和 P_j 数据不相关 (间接相关除外)。以上的定义也适用于语句块和循环中的语句。

这三种数据相关情况对程序并行性的影响表现为下列几种可能的执行次序:

(1) 写-读串行次序

如果程序必须保持在先的语句先写、在后的语句后读的次序，则允许存在上述任一种数据相关情形。一般情况下，执行的次序不可并行，也不可颠倒。这是通常遇到的典型串程序序。但有一种特殊情形，即当 P_1 和 P_2 服从交换律时，虽仍需串行执行，但允许 P_1 和 P_2 执行的次序对换，这称为可交换串行。例如

$$P_1 \quad A = 2 * A$$

$$P_2 \quad A = 3 * A$$

为可交换串行；但

$$P_1 \quad A = B + 1$$

$$P_2 \quad B = A + 1$$

就是不可交换串行的。

(2) 读-写次序

如果二程序段之间只包含第二种数据相关情形，则只需保持在先的语句先读、在后的语句后写的次序，便能既允许它们串行，也允许它们并行执行，但不允许交换次序。例如

$$P_1 \quad A = B + D$$

$$P_2 \quad B = C + E$$

二者同时执行，能保证 P_1 先读 B ， P_2 后写 B 的次序。又如果三个处理机共享存贮器，同时分别执行下列 3 个语句：

$$P_1 \quad A = 3 * C / B$$

$$P_2 \quad B = C * 5$$

$$P_3 \quad C = 7 + E$$

虽属可能，但由于 P_1 费时最长， P_2 次之， P_3 最短，故如果不采取特别的同步措施，就不能保证必要的先读后写次序。图 7.38 所示的多处理机结构形式对执行读-写次序的并行程序比较优越，它允许每个处理机的操作结果暂时存放在自己的本地存贮器内，而不急于去修改原来存放在共享存贮器中的单元内容。这样只要控制本地存贮器向共享存贮器的写入同步就够了。

(3) 可并行次序

如果程序的两段之间不存在任一种数据相关情况，即无共同变量，或共同变量都在右边部分，且不在左边部分出现，则二者可以无条件地并行执行。当然也可以串行执行，而串行时可以选择任一先后顺序。例如

$$P_1 \quad A = B + C$$

$$P_2 \quad D = B + E$$

(4) 必并行次序

如果二程序段的输入变量互为输出变量，则二者必须并行执行，而不允许串行执行。例如，二语句的左、右变量互相交换

$$P_1 \quad A = B$$

$$P_2 \quad B = A$$

必须并行执行，且需保持读写完全同步。当然，如果在图 7.38 的多处理机上运行这样的程序段，便能自动保证读-写次序，降低对同步的要求。

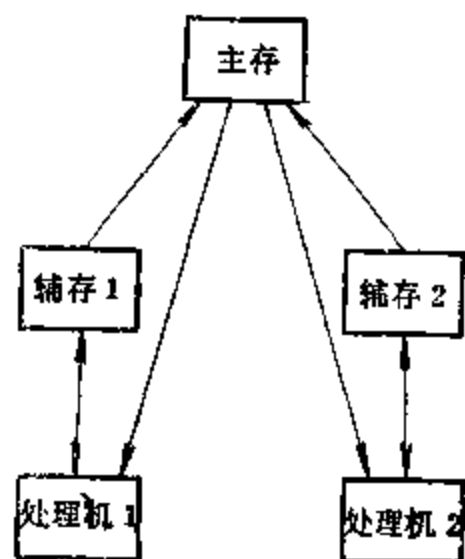


图 7.38 能保证读-写次序的多处理机结构

以上分析的五种执行次序分属串行和并行两大类，一部分是二者的重迭，其相互关系用图解法表示在图 7.39 中。按照可交换和不可交换可将串行程序分为两类。再按照可并行和不可并行又将每一类分为两种。全部五种执行次序包含了所有可能的情形。程序中还需要有专门的语句或指令来具体描述这些并行关系。这将在下一小节叙述。

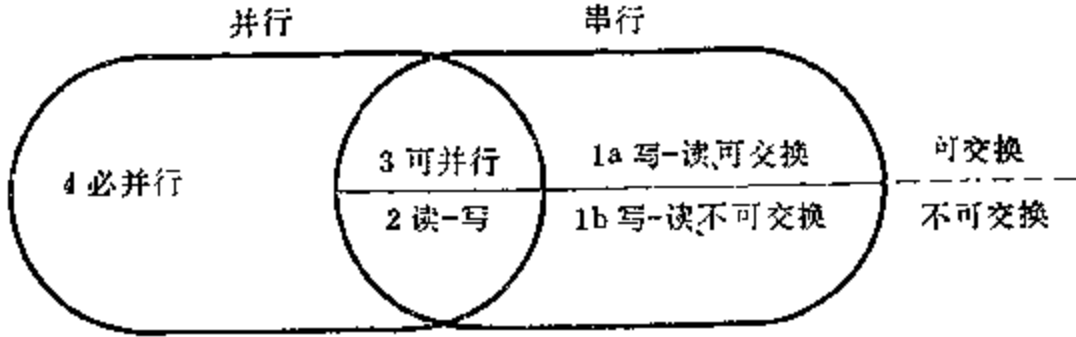


图 7.39 五种不同执行次序的图解表示

3.4 并行进程的控制和调度

包含并行性的程序在多处理机上运行时，需要有相应的控制机构来实现管理功能，这包括并行任务的派生、同步和调度，主要是通过操作系统用软件手段来实现的。由于许多内容在操作系统课本中已有叙述，本节仅结合多处理机的特点择要介绍。

3.4-1 并行任务的派生与汇合

并行任务的派生就是使一个任务在执行的同时派生出可与它并行执行的其它一个或多个任务，分配给不同的处理机完成。这些任务可以是互不相同的，执行时间也各异，要等它们在先后不同时刻全部完成以后，再汇合起来，进入后继的任务。这后继的任务既可以是单任务，也可以是新的并行任务。如果是并行任务，那么又要继续派生，继续汇合，……依此进行下去，直至整个程序结束。

并行任务的派生和汇合通常是用软件手段来控制的，首先需要在程序中反映出并行任务的派生和汇合关系。FORK和JOIN语句是普遍采用的一种方法。

先看一个用多处理机计算式 (7.3-6) Z 值的例子，并行程序已在 § 3.3-1 节中得到了，现在重写如下：

```
S1  G = A * B
S2  H = C / D
S3  I = G * H
S4  J = E + F
S5  Z = I + J
```

如果不加并行控制指令，这个程序只是一个普通的串行程序，不能发挥多处理机的作用。图 7.40 示出了各指令之间的数据相关情况，它表明 S₁ 和 S₂ 可以同时开始执行，但要等到 S₁ 和 S₂ 都完成以后，才能开始执行 S₃；并行地可以执行 S₄，而 S₄ 和 S₃ 汇合才能执行 S₅。实现这个派生和汇合关系的程序如下：

```
      FORK  S2
S1  G = A * B
      JOIN  2
```

```

        GOTO S3
S2  H = C/D
        JOIN 2
S3  FORK S4
      I = G*H
        JOIN 2
        GOTO S5
S4  J = E + F
        JOIN 2
S5  Z = I + J

```

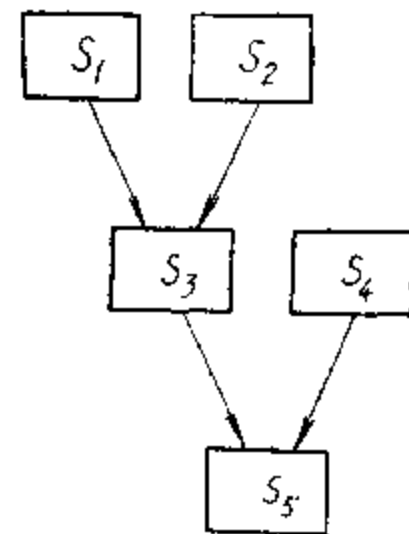


图 7.40 计算式 (7.3-6) 的并行程序数据相关图

执行这个程序可用 2 个处理机。假定最初的程序是在处理机 1 上运行的, 遇到 FORK S₂ 指令时, 就分出一个处理机 (假定是 2) 去执行 S₂, 而处理机 1 接着执行下面的 S₁ 指令。S₁ 执行时间较短, 当它结束时, S₂ 尚在执行中, 遇 JOIN 指令时, 处理机 1 从 S₁ 释放, 如果没有其它任务, 则释放后处于空闲。随后, 当 S₂ 结束时, 由于已与 S₁ 汇合, 便可以通过 JOIN 指令, 由处理机 2 继续执行后续的 FORK S₄ 指令。这条指令又派生出 S₄, 分配给空闲的处理机 1, 而处理机 2 接着执行 S₃。同样, 要等 S₃ 和 S₄ 都先后结束, 才满足 JOIN 指令的汇合条件, 进入指令 S₅。这个执行过程示于图 7.41 的资源时间图中。

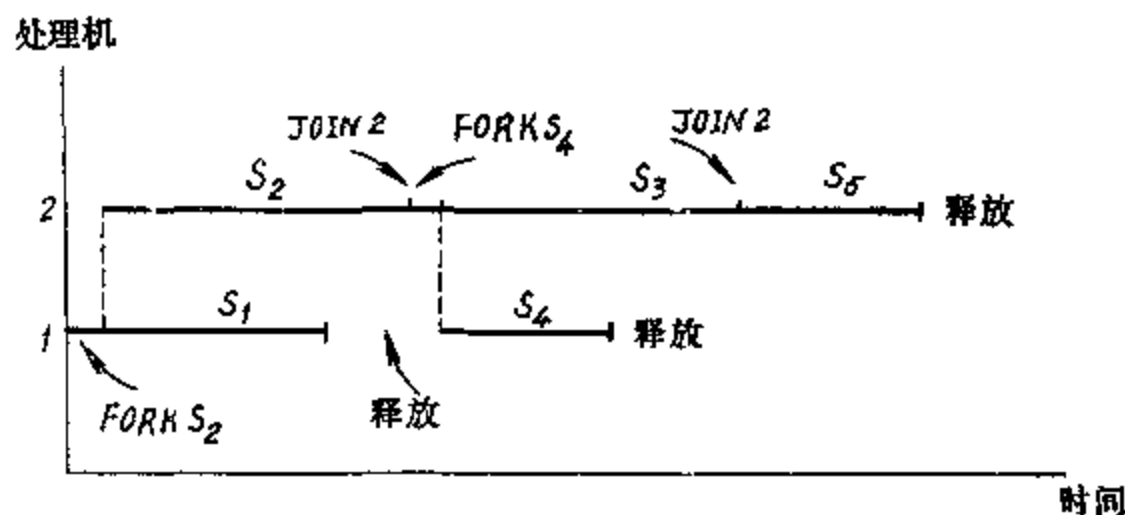


图 7.41 式 (7.3-6) 计算程序在多台处理机上运行的资源时间图

下面归纳说明 FORK 和 JOIN 指令的功能。关于它的细节可能会有不同的方案^[31-6], 我们选用 M·E·康维 (Conway) 最早建议的一种。

关于 FORK 指令:

(1) 在遇到 FORK 指令时, 执行这条指令的原进程, 根据 FORK 指令所附的标记符派生出该标记符所对应的新进程。具体地说, 就是准备好这个新进程启动和继续执行所必需的有关信息, 如果是共享内存, 则应该为它产生存储器指针、映象函数和访问权等数据。

(2) 执行 FORK 指令的原进程, 继续在分配给它的处理机上运行。

(3) 将空闲的处理机分配给被 FORK 指令派生的新进程; 如果没有可用的处理机, 则让它排队等待。

与 FORK 指令相配合, 规定 JOIN 指令的功能如下:

(1) JOIN指令附有一个计数器，其初始值置为0。当执行 JOIN N指令时，计数器的值加1，并与N比较。

(2) 若比较结果表明计数器的值等于N，则说明这是执行中的第N个进程经过 JOIN 指令，于是允许该进程通过 JOIN指令，在其所在的处理机上继续执行后续指令。

(3) 若计数器的值小于N，则必须等待N个并行任务中尚未执行或虽执行但未结束的进程达到JOIN指令。现正执行JOIN指令的这个进程可先结束，把它占用的处理机释放出来，分配给正在排队等待的任务。

上面的例子只是为了说明FORK和JOIN指令的意义。用它们单纯实现指令之间的并行是不切实际的，因为增加的指令开销早就超过了受益。所以我们在了解了FORK和JOIN语句的功能以后，还是来看一个在多处理机上求解矩阵乘问题的程序，以便与上节为并行处理机写的同一题目的程序进行比较。

仍假定A, B两个 8×8 矩阵相乘，需要在多处理机上实现任务一级即外循环的并行，用FORTRAN语言书写的程序如下：

```

        DO 10 FOR J=0,6
10    FORK 20
        J=7
20    DO 30 I=0,7
        C(I,J)=0
        DO 40 K=0,7
40    C(I,J)=C(I,J)+A(I,K)*B(K,J)
30    CONTINUE
        JOIN 8

```

假定执行FORK指令的进程是在处理机1上运行。在J依次等于0至6的情形下，连续执行7次 FORK 20 指令，派生7个以20为标记符的进程，提出了让它们与自己接着往下执行的J=7的同一进程相并行的要求。如果处理机数目只有3个，分配给J=0和J=1的进程以后，其余J=2,3,...,6的5个进程就得排队等待，而原处理机1在离开FORK循环以后进入J=7的进程。整个程序在陆续执行完全部8个进程时才告结束，其资源时间图如图7.42所示。

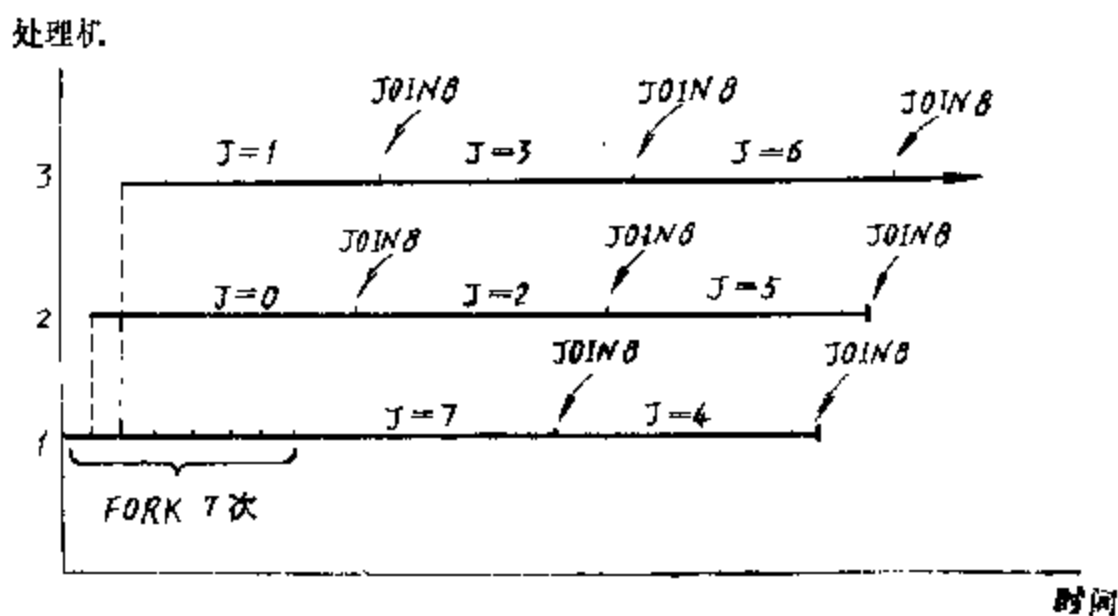


图 7.42 矩阵乘程序在多处理机上运行的资源时间图

从表面上看，为了求解同一矩阵乘的题目，多处理机的每一个处理机和并行处理机的每一个处理单元完成的工作是一样的，但是这两种处理方式是有根本区别的：第一，并行处理机的每一条指令要求 8 个处理单元完全同步地对 $J=1, \dots, 7$ 的不同数组进行运算，而在多处理机中，即使有 8 个处理机执行同一程序段，并不需要、也不会完全同步；何况一般说起来，不同处理机所执行的程序段可以是毫不相同的。这就是操作级并行与任务级并行的差别。第二，多处理机中可用的处理机数目对程序的书写没有影响，换言之，程序对可用的处理机数目并无固定要求。这是因为，处理机的分配和释放都是由操作系统自动控制的，它们已经反映在 FORK 和 JOIN 指令的功能中。这是多处理机相对并行处理机的重要优点之一。

3.4-2 同步与互斥

并行任务之间同步的必要性在 § 3.3-3 节讨论程序执行次序时已经提到了。这里再举一个能最好说明两个并行任务之间同步必要性的例子。它包含被称为“生产者”和“消费者”两个相关的并行循环过程，如图 7.43 所示。过程名称的由来是这样的：假定“生产者”制作某一种食品，每生产一个就放置一个在只有 N 个位置的烘箱内，供顾客随时来买。而“消费者”即顾客每次从烘箱内取走一个食品，随即消费一个。只有当烘箱内存货不为零时，顾客才能有货可取；同样，只有当烘箱内存货不大于也不等于 N 时，生产者才能继续制作而不致无处可放。不满足这两个条件之一时，相应的过程就必须等待，而让另一过程继续进行到使造成停产或缺货的局面得到扭转。这就提出了两个过程之间的同步问题。

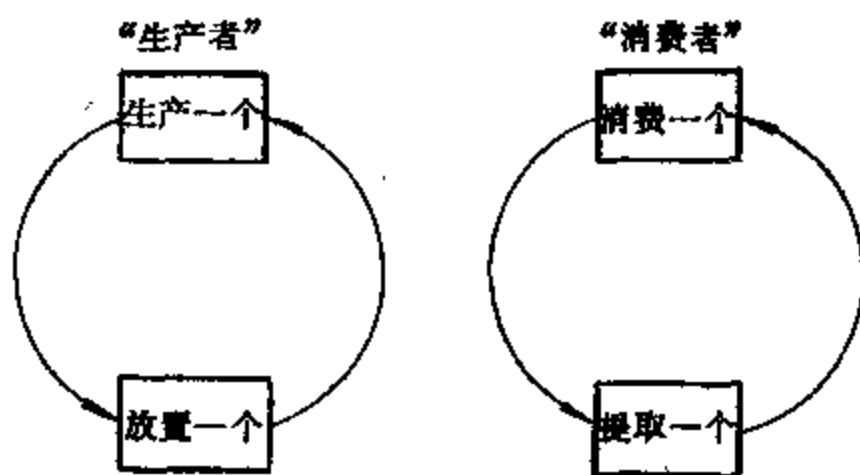


图 7.43 “生产者”和“消费者”的同步要求

类似的过程也发生在计算机中，例如“生产者”可以是卡片读入机，读入的数据放置在一定容量的缓冲器中，“消费者”可以是宽行打印机，从缓冲器内取出数据打印。所以过程同步问题即使在单处理机的 I/O 过程与中央处理器过程之间也是存在的。

解决同步问题有多种方案，基本原理是设计两条互相制约的指令。E·M·狄克斯特拉 (Dijkstra) 最早称它们为 $P(s)$ 和 $V(s)$ ，它们对一些特殊的公共变量进行处理，并根据这些变量的取值情况控制有关过程间的同步。这些公共变量的特殊性在于：第一，它们只能取整数值，每次只能被一个过程进行加 1 或减 1 处理，而且处理期间不得被另外的过程所打断；第二，它们除了被规定的 P 和 V 指令处理外，不得为任何其它的操作所改变。狄克斯特拉把这种变量称为信号灯 (Semaphore)。

将 P 和 V 指令分散放于两个需要同步的循环过程中²⁹，如图 7.44 所示。一对 P, V 指令的公共变量为 e ，代表库空量，起始值为缓冲器的最大库容量 N ；另一对 P, V 指令的公共变量为 f ，代表库存量，起始值为零。在“生产者”循环中，每一次 $P(e)$ 对 e 减值并测试。当减到小于零时，表示缓冲器已满， $P(e)$ 控制“生产者”暂停，等待“消费者”循环经过 $V(e)$ 指令对 e 增值。在“消费者”循环中，每一次 $P(f)$ 对 f 减值并测试。当减到小于零时，表示缓冲器已空， $P(f)$ 控制“消费者”暂停，等待“生产者”循环经过 $V(f)$ 对 f 增值。

狄克斯特拉对 $P(s)$ 和 $V(s)$ 所作的规定如下。

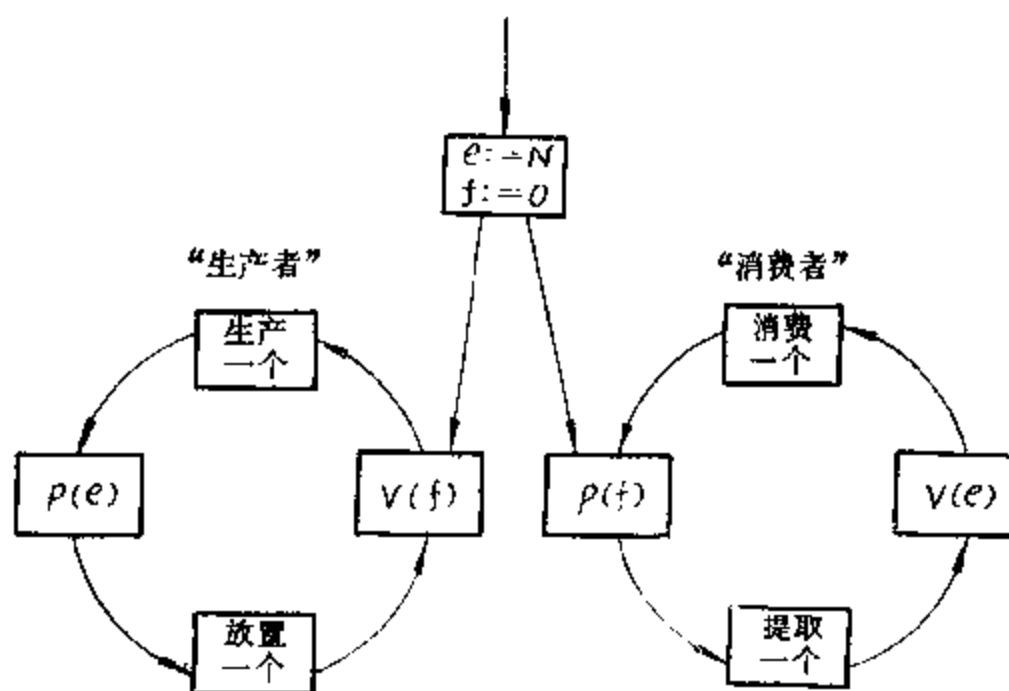


图 7.44 “生产者”与“消费者”的同步实现

对 $P(s)$:

- (1) 将 S 减值, 即 $S := S - 1$;
- (2) 若 S 等于非负值, 则 $P(s)$ 所在的原进程可以继续执行其后的语句;
- (3) 若 S 为负, 则原进程暂停, 保存该进程状态, 将它列入一个与 S 相联的等待排队站中。至少在这个 S 被 $V(s)$ 增值以前, 对该进程都不给分配处理机。

对 $V(s)$:

- (1) 将 S 增值, 即 $S := S + 1$;
- (2) 若 S 的结果为正, 则 $V(s)$ 所在的进程不受任何影响, 继续执行;
- (3) 若 S 的结果等于非正值, 则从与 S 相联的等待排队站中, 取出一个等待的进程, 列入准备就绪排队站, 以便在适当时刻分配给它空闲的处理机。
- (4) $V(s)$ 所在的原进程继续执行。

$P(s)$ 和 $V(s)$ 也可用于多个进程之间的同步。允许信号灯 S 取负值。 $S = -n$ 表示有 n 个进程因执行 $P(s)$ 而被列入了等待该 S 的排队站中。如果 $V(s)$ 增值后获得的 S 结果值仍为负, 说明与之相联的等待排队站中已经包含了不止一个进程, 这时, 具体移出哪一个进程是无关紧要的。

P 和 V 指令除了用于进程同步以外, 还可用于互斥, 以便保护某一关键程序段在同一时间内只能被一个进程执行, 而排斥掉任何其它想要执行该程序段的进程, 让它们处于等待状态。服务于互斥目的的 P 和 S 指令用法如下:

置 S 初始值为 1;

在进程 1 中

```

:
:
P(s)
关键程序段
V(s)
:

```

在进程 2 中

⋮
P(s)
关键程序段
V(s)
⋮

在用于多个并发进程的互斥时，S 的最大值等于 1，最小值等于 $-(n-1)$ ，此处 n 为并发进程的个数。

互斥在多处理机的进程控制中也是必要的。例如几个处理机并行执行下一程序：

```
DO 10 I = 1, N  
10 SUM = SUM + X(I)
```

每一个处理机执行读 SUM，加 X(I) 和存 SUM 几个步骤必须一气呵成，不允许被其它处理机从中打断和插入，否则所形成的 SUM 结果就会面目全非。办法之一就是利用 P, V 指令，写成

```
S = 1  
P(s)  
SUM = SUM + X(I) (关键程序段)  
V(s)
```

只有在 S 减值后得 $S = 0$ ，才允许经过 P(s) 进入关键程序段；在出口处，V(s) 又恢复 $S = 1$ ，允许其它进程进入。若在 P, V 之间遇到另一进程试图经过 P(s) 进入，则 $S = 0$ 减值后变为 -1 ，该进程被挂起。等到前一进程执行完毕，经过 V(s) 将 S 增值为 0，唤醒被挂起的进程进入关键程序段。前一进程经过 V(s) 后继续往下执行。

在计算机中具体实现 P 和 V 操作，较合理的方法是采取操作系统原语的形式，在所需条件满足时，通过陷阱指令转入操作系统调用之。为了使一个进程的增值或减值操作不被其它进程所打断，在指令系统中设置 INCREMENT AND TRAP、DECREMENT AND TRAP 或 TEST AND SET 等类指令是很有用的。

3.4-3 资源分配和进程调度

资源分配和进程调度是决定多处理机效能的关键问题。这个问题在单处理机的分时和多道程序运行中也是要遇到的，但是在多处理机中却变得更为复杂，特别是在处理机数目很多的时候。

多处理机中存在多个实际的处理机是它区别于单处理机分时系统的主要特点之一。因此，对于多处理机而言，资源分配要解决的根本问题是处理机分配；而把处理机分配给提出要求的进程便是进程调度的主要内容。为此，非对称型多处理机进行资源分配要根据功能专用化和功能分布的原则；这就是说，把控制功能、I/O 功能、处理功能（各种语言的解释和编译、机器仿真、应用程序运行、数据库管理等）分配给专门的处理机完成，而为了在信息流量变化的情况下保持处理机的负载平衡，要进行动态分配。对称型多处理机进行资源分配，要根据资源池原则；就是说，多种资源，包括处理机、内存、I/O 通道等，放在一个公共的资源池里，由多个进程共享，以取得设备的高使用效率和处理机的负载平衡。

首先介绍几种控制功能分配方案，它们是专用控制、分布控制和浮动控制。专用控制是固定设置一台专门的控制处理机，实现整个系统的集中控制。这种系统的优点是简单、经济、快速和便于用硬件或固件实现进程调度，但要求控制处理机具有很高的工作可靠性和优越的性能指标，才能对一个有相当规模的系统实现有效的控制。分布控制则与集中控制相反，是把控制功能分散给多台处理机，而且每一处理机都有可能被分配承担控制任务。在同一时刻处于控制状态的几台处理机共同分担整个系统的控制工作。这种系统的优点是更适应分布处理的模块化结构特点，减少对大型控制专用处理机的需求，能保证较高的工作可靠性，以及把控制进程和用户进程一起进行调度，能取得较高的系统利用率。但是，这种进程调度必然会复杂得多。例如，至少是操作系统的内核应重复多套。当某一处理机提出了控制要求时，应分配运行进程优先级最低、最易被中断的处理机来处理这一控制任务。这样，对操作系统共享部分的访问冲突势将增多，也会使调度的复杂性和开销加大。浮动控制则是上面两种控制方式间的折衷方式。它在一段较长的时间内指定某一个处理机为控制处理机，但是具体指定哪一个处理机以及担任多长时间都不是固定的。这种系统在硬件结构和可靠性上具有分布控制的优点，而在操作系统的复杂性和经济性上则接近专用控制。

进程控制主要指进程切换。为了提高处理机使用效率，当任一进程由于某种原因（等待某种外部条件满足或转到 I/O 或其它类型专用处理机）而不能继续进行，要求将处理机释放，以便重新分配给其它等待中的进程使用。这就要求改变进程状态，称为进程切换。关于进程状态，在不同的多处理机中，可以有不同的规定。粗略说来，最基本的进程状态有两种，即运行状态和挂起状态。当进程获得了处理机和其它资源而正在执行指令时，称进程处于运行或执行状态。但在需要等待某种外部条件满足而无有用事情可做的时候，就要转到封锁或挂起状态，在这一状态下，进程要把它占用的处理机释放出来，而且为了防止死锁，有些系统还要求它不再占用内存和其它设备。为了在一个进程被正式挂起以前先检查它是否占有这些设备，以及是否正在向/从内存送进/出信息，需要先经过一个中间过渡状态，以便执行一个“事件检查”（TEST-EVENT）

操作，避免正在传送的信息进入错误的地址，这时称进程处于等待状态。已经挂起的进程，如遇到外部条件满足，在重回运行状态以前，也要先经过一个中间过渡状态，以便排队等待分配处理机，这时称进程处于就绪状态。这四种状态之间的切换关系示于图 7.45 中。

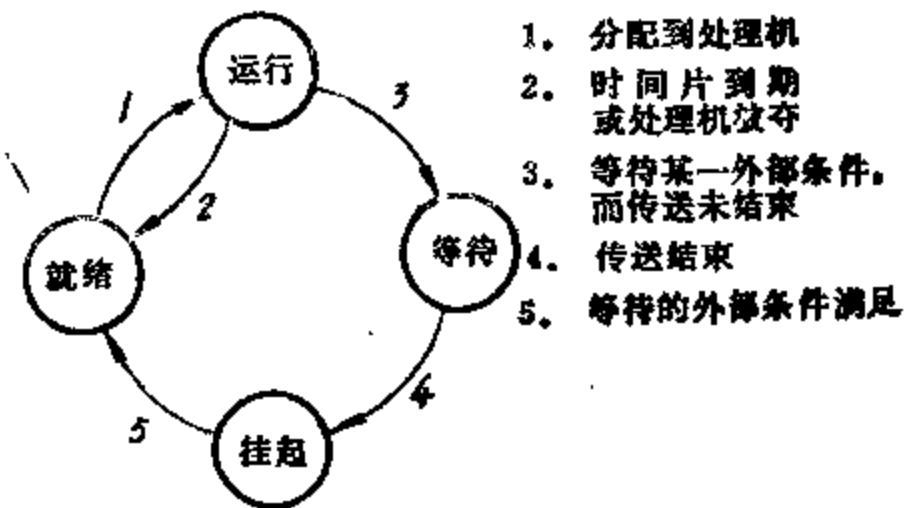


图 7.45 四种进程状态之间的切换关系

进程由一个状态转为另一个状态，需要有信号控制，这称为进程操作，表现为有关的操作系统原语，它们是调度程序的主要组成部分。举 EPOS 多处理机为例，这些操作系统原语可以是 SUSPEND-PROCESS, RESUME-PROCESS, SEND, RECEIVE, TEST-EVENT 等。当处于运行状态的进程需要封锁时，由它自己或由其它进程向调度程序发出封锁请求，在操作系统原语 SUSPEND-PROCESS 的控制下，执行 TEST-EVENT 原语。如果测试结果表明还有未结束的信息传送正在进行，则过渡到等待状态。TEST-EVENT 的功能是单纯测试一个公共变量但不改变它的值。这个公共变量就是用来实现信息传送互斥

的信号灯，只能由 SEND 和 RECEIVE 原语分别减值和增值，其原理就象前面所述 P 和 V 操作一样。信息传送结束时，信号灯增值，允许进程从等待状态离开，进入封锁状态。这时，要保留现场和释放资源，把原进程的状态向量，包括程序状态字、中央寄存器内容、地址空间及内容、I/O 设备状态等保存起来。释放出来的处理机从就绪排队站中寻找新进程，取回新的进程状态向量，继续执行；若找不到，则暂时进入空闲状态。注意：当一个进程被封锁以后，它自己就不再能发出“苏醒”信号，因此它必须在进入封锁状态前作好安排，例如依靠另一进程对某一公共变量增值或让某种 I/O 服务程序结束时产生这一信号等。“苏醒”信号利用操作系统原语 RESUME-PROCESS 进行控制，将被封锁的进程转入就绪状态，排队等待分配处理机。

要求进程被封锁前释放所有占用的资源，这是防止死锁的一种办法，但过分增加了状态切换的开销。如果允许处于封锁状态的进程保留除处理机以外的其它资源，则必须采取其它有效的措施防止死锁。下面是进程被封锁后仍占用资源可能造成死锁的例子。

假定进程 A 和 B 在第一阶段各占用 $1/2$ 的内存容量，接着又同时请求各追加 $1/4$ 的内存容量。从每个进程来讲， $3/4$ 的容量要求是允许的，不超过总的限度，但是在运行过程中就会发生问题。试看下列运行过程：

- (1) 进程 A 要求和被给予 $1/2$ 内存；
- (2) 进程 B 要求和被给予 $1/2$ 内存；
- (3) 进程 A 要求追加 $1/4$ 内存，不能满足，被挂起，它已占用的 $1/2$ 内存也不释放；
- (4) 进程 B 要求追加 $1/4$ 内存，不能满足，被挂起，它已占用的 $1/2$ 的内存也不释放。

于是陷入了僵局，进程 A 和 B 互相等待，但都不能等到对方结束后获得新的内存补充。整个系统进入死锁状态，不能继续完成任何有用的工作。

分析造成死锁的原因，可知下面三点合起来构成死锁的必要条件：

- (1) 进程排它性地占有某些系统资源；
- (2) 当进程对资源的进一步要求被拒绝而挂起时，继续占有已用的资源而不释放；
- (3) 资源占有状况出现死循环，即 A_1 要求的资源被 A_2 占有， A_2 要求的资源被 A_3 占有……，如此类推，最后 A_n 要求的资源又被 A_1 占有。

防止死锁的方法就是从任一环节上打破死循环，使上述条件至少有一个不能满足，例如：

(1) 进程被挂起后不能继续保持对资源的控制，或者说，强制即将被挂起的进程放弃已占用的资源，待苏醒时再重新申请。

(2) 进程必须一次提出对全部所需资源的申请，在要求未能满足以前，不能占有任何资源，一旦满足，则在它整个运行期间都保留它们而不释放。

(3) 在进程要求多种资源的情况下，对资源规定一个固定的次序，各个进程都必须按照这个先后次序提出资源申请，可以避免几个进程相互要求资源的死循环。

(4) 把系统可用的各种资源数、各个进程一次需用的最大资源数，以及每个时刻实际分配的资源数都列成表。当每次有进程提出资源申请时，都要由操作系统核算一下，看看如果满足后任一进程再申请资源达到它预先登记的最大数时是否会造成死锁。

防止死锁是一个全局性问题，虽然很重要，但还不是多处理机资源管理问题的全部。还必须进一步从保证系统高效率的角度来处理资源分配和进程调度问题，而且依据的算法应该

是快速易行的。目前这个问题还只是在少数极度受限制的情形中得到解决。通常在多道程序中行之有效的进程调度方法,如优先级调度、时限调度等,都不完全适用于以任务级并行为特征的多处理机,这是因为任务之间有很复杂的内在联系,它们的调度策略不易用优先级等简单准则来表示;或者说,如果仍用优先级控制,那么如何来指定优先级,问题便又回到了用户手中。寻找更合理、快速、高效的进程调度算法是当前多处理机操作系统的重要研究课题之一,只有这个问题与并行算法问题一起得到较实际的解决,包含处理机数目很大的多处理机系统才有可能成为现实,获得象并行处理机那样成功的应用。

主要参考文献

- [1] David J. Kuck, "The Structure of Computers and Computations," vol. I, John Wiley & Sons, 1978.
- [2] P. H. Enslow (ed.), "Multiprocessors and Parallel Processing," John Wiley & Sons, 1974.
- [3] Harold Lorin, "Parallelism in Hardware and Software, Real and Apparent Concurrency," Prentice-Hall Inc, 1972.
- [4] K. J. Thurber, "Large Scale Computer Architecture, Parallel and Associative Processors," Hayden Book Co. Inc, 1976.
- [5] J. P. Hayes, "Computer Architecture and Organization," McGraw-Hill Book Co, 1978.
- [6] H. S. Stone (ed.) "Introduction to Computer Architecture," Science Research Associates, 1975.
- [7] 金兰等, "并行处理计算机结构," 国防工业出版社, 1982.
- [8] D. J. Kuck, "A Survey of parallel Machine Organization and Programming," Computing Surveys, vol. 9, No.1, March 1977, pp.29-59.
- [9] P. H. Enslow, Jr., "Multiprocessor Organization-A Survey," Computing Surveys, vol. 9, No.1, March 1977, pp. 103-129.
- [10] S. S. Yau and H. S. Fung, "Associative Processor Architecture-A Survey," Computing Surveys, vol. 9, No.1, March 1977, pp. 3-27.
- [11] C. V. Ramamoorthy and H. F. Li, "pipeline Architecture," Computing Surveys, vol.9, No.1, March 1977, pp.61-102.
- [12] P. H. Enslow, Jr., "What is a 'Distributed' Data Processing System?" Computer, vol. 11, No.1, Jan. 1978, pp. 13-21.
- [13] K. J. Thurber and H. A. Freeman, "Architecture Considerations For Local Computer Networks," Proc. of the 1st Conference on Distributed Computing Systems, Oct. 1-5, 1979, pp. 131-142.
- [14] M. J. Flynn, "Very High-Speed Computing Systems," Proc. IEEE, vol. 54, No. 12, Dec. 1966, pp. 1901-1909.
- [15] W. J. Bouknight, et al, "The ILLIAC IV System," Proc. of the IEEE, vol. 60, No.4, Apr. 1972, pp. 369-388.
- [16] Lai-wo Fung, "A Massively Parallel Processing Computer," High Speed Computer and Algorithm Organization, ed. D. J. Kuck et al., Academic Press, 1977,

pp. 203-204.

- [17] P. M. Flanders, et al, "Efficient High Speed Computing with the Distributed Array Processor," High Speed Computer and Algorithm Organization, ed. D. J. Kuck et al., Academic Press, 1977, pp. 113-128.
- [18] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)".
- [19] H. J. Siegel, "Interconnection networks for SIMD machines," Computer, vol. 12, No.6, June 1979, pp.57-65.
- [20] K. E. Batcher, "The Flip Network in STARAN," Proc. of the 1976 International Conference on Parallel Processing, pp. 65-71.
- [21] M. C. Pease, "The Indirect Binary n-cube Microprocessor Array," IEEE Trans. on Computers, vol. C-26, No.5, May 1977, pp. 548-573.
- [22] H. S. Stone, "Parallel Processing and the Perfect Shuffle," IEEE Trans. on Computers, vol. c-20, Feb. 1971, pp. 153-161.
- [23] T. Feng, "Data Manipulating Functions in Parallel Processors and Their Implementations," IEEE Trans. on Computers, vol. c-23, Mar. 1974, pp. 309-318.
- [24] S. H. Fuller, et al, "Multi-Microprocessors, An Overview and Working Example," Proc. IEEE, vol. 66, No.2, Feb. 1978, pp. 216-228.
- [25] D. A. Patterson, E. S. Fehr, and C. H. Sequin, "Design Considerations for the VLSI Processor of X-TREE," The 6th Annual Symposium on Computer Architecture, 1979, pp. 90-101.
- [26] Jin Lan, "A New General-purpose Distributed Multiprocessor System Structure," Proc. of the 1980 International Conference on Parallel Processing, pp. 153-154.
- [27] S. E. Madnick and J. J. Donovan, "Operating Systems," McGraw-Hill Co. 1974.
- [28] A. J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE Trans. on Electronic Computers, vol. EC-15, No. 5, Oct. 1966, pp. 757-763.
- [29] N. Wirth, "On Multiprogramming Machine Coding, and Computer Organization," Comm. of the ACM, vol. 12, No.9, Sep. 1969, pp. 489-498.
- [30] A. N. Habermann, "Prevention of System Deadlocks," Comm. of the ACM, vol. 12, No.7, July 1969, pp. 373-377, 385.

第八章 描述与评价

“描述”指的是硬件的描述语言，“评价”指的是计算机系统的性能评价，它们都是研究和设计系统结构所必不可少的。

电子计算机大大推动了很多学科的发展，不论理论上还是实践上都是如此，尤其是自动化的发展，自动控制系统、自动化工厂或车间，其核心就是计算机，因此人们也称计算机为电脑。然而在研究和设计计算机本身方面，这个电脑起的作用却远不如它在其它学科的理论研究和在自动控制系统中所起的作用大。目前，不论是硬件的设计，还是软件的设计，科学性与自动化程度都较低。不要说从无到有设计出整个复杂的计算机系统需要化费多少年，就是将已经设计好的图纸和文字说明校对一遍也是相当费时费事的。即便想了解全部设计结果，也并非易事，更谈不上为要确认该设计方案是正确可行的又需要耗费多少人力和财力。人们迫切希望简化计算机系统的设计过程。普遍认为，目前，计算机这个领域还未真正进入“科学”阶段，而仍处于“技艺(Art)”阶段。

人们期望有朝一日整个计算机系统（包括软件与硬件）能自动生成，自动设计（或选用）。设计者只需要给出设计要求，就能由计算机构成的自动设计系统最优地设计出其软件和硬件。就硬件来看，其自动设计可设想成如图 8.1 所示的过程。

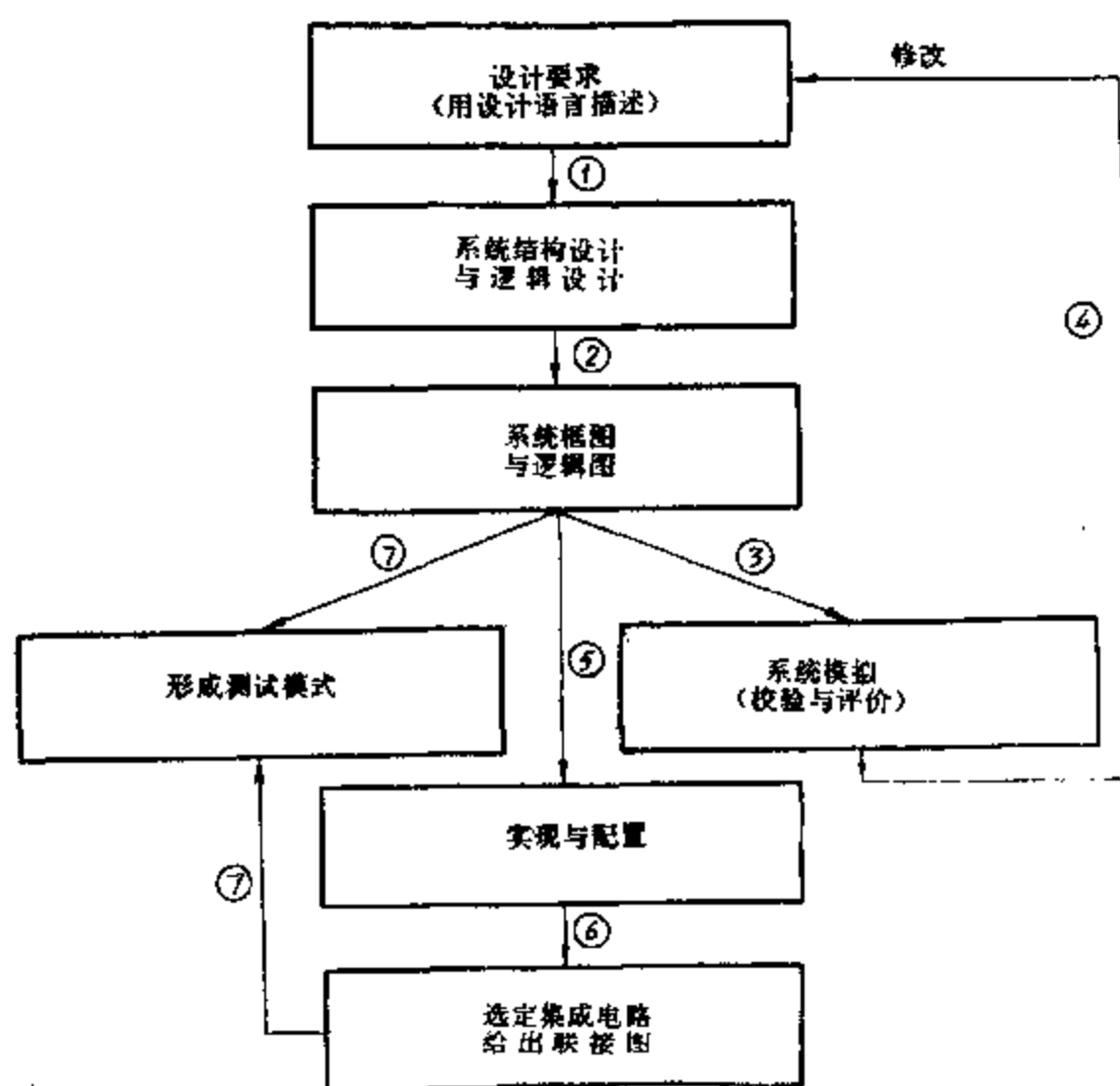


图 8.1 计算机的自动设计

为此，第一步必须要有简明的描述语言，用来说明设计要求并送入自动设计系统；然后由自动设计系统进行系统结构设计及逻辑设计并给出系统框图与逻辑图；再由系统模拟软件

对它们进行校验（包括正确性检查和指出在哪里有设计错误）和性能评价；而后由模拟软件修改设计语句。这个过程的顺序用图 8.1 中①、②、③、④表示，它具有某种自探索能力，能够通过多次循环（包括设计人员的参予），给出性能最优而正确的设计，这个过程是自动设计的核心，也是最难实现的。

在设计好系统框图与整机逻辑图后，就要进行实现设计（包括微程序的自动设计）和配置设计（如部件的划分，插件的划分等）；然后选定集成电路，给出插件内及插件间的联线图；最后，要自动形成从插件直至各部件的测试、诊断模式。这个过程的顺序用图 8.1 中⑤、⑥、⑦表示。至于印制板的自动制板以及按设计好的诊断模式进行自动测试，在七十年代已经基本解决。

为了能达到这种自动设计，有大量理论上和实践上的问题需要解决。其中，下述二方面必须解决，一是硬件的描述语言，二是计算机系统的性能评价。要能够实现由计算机执行的自动设计就必须首先解决从整机到部件直至组件的描述，之后才能按一定算法编成程序。要能编制出设计机器的程序，设计出好的机器，首先又得解决什么叫好，什么叫坏，即性能评价问题。这二方面是基础，它们不解决，其它就谈不上。当然，研究它们不只是为了计算机自动设计，这卅年来，在这二方面进行了大量的工作，它们有着广泛的用途。

我们先讲讲硬件描述语言。

§ 1 硬件描述语言

1.1 什么是硬件描述语言

硬件描述语言是用来描述计算机硬件组成及其操作的一种语言。硬件描述语言应能描述硬件系统的各个方面。诸如算法，指令系统，处理机、存储器、外部设备和开关网络等硬件的组成，寄存器间的操作以及门与触发器表示的网络等。

有了硬件描述语言所带来的好处是：

- (1) 计算机工程人员之间，以致软件和硬件人员之间有了方便的交流工具；
- (2) 便于给出精确、简明的设计和文件资料；
- (3) 可在计算机上进行模拟校验，从而可使绝大多数错误不必在制造出机器后才能被发现；
- (4) 有助于实现包括线路设计、装配设计、逻辑设计、结构设计、性能设计及系统设计等方面的自动化。

实际上，自从有电子计算机后就有硬件描述语言，否则无从给出计算机的图纸与说明。

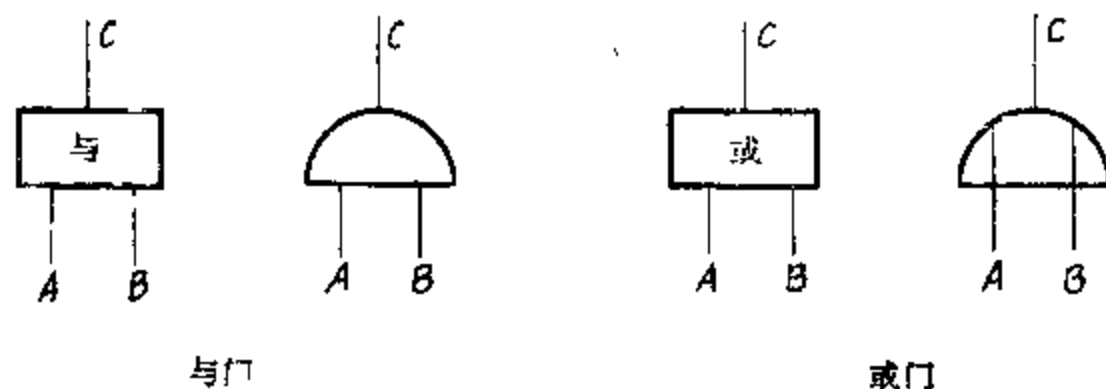


图 8.2 描述与、或门的图形式语言

例如,对计算机的基本单元:与、或门等,在五十年代就已有各种图形描述方法,如图8.2那样。此外,还可以用布尔代数来描述,如:

与门 $C = A \wedge B$

或门 $C = A \vee B$

这些,大家都很熟悉。可见,硬件的描述语言有图形式语言(如图8.2所示的逻辑图语言)和文本式语言(如布尔代数语言)二类,它们各有其优缺点。

虽然逻辑图语言和布尔代数语言至今还在广泛应用,但是它们毕竟有很大的局限性。例如,布尔代数语言适合于描述一般组合网络,对有复杂时间关系的网络就难于胜任,要用它描述大部件、指令系统或整机就更为困难,甚至不可能。又如逻辑图语言,在门及寄存器级,它能精确描述,对有复杂时间关系的部件与整机等,用图8.2的与、或门级的逻辑图语言表示,就会极为复杂,因此一般是用逻辑框图来粗略描述,仅给出示意图。逻辑图语言的更大缺点还在于它只能描述逻辑设计的结果,不能描述逻辑设计的要求;还有,它几乎不能用于计算机的输入。到了六十年代,随着计算机的日益膨大、复杂,人们就要求能在硬件系统实际构成之前,先用计算机对它进行模拟,检验其正确性,而用逻辑图语言描述的硬件系统逻辑图是不能直接送入计算机进行模拟的。

因此从六十年代开始,随着高级程序设计语言的广泛应用和完善,人们就研究能否引用软件在语言上的成果来描述硬件。一种途径是直接引用,例如,直接用APL或PL/I语言描述,其最大好处是可以使用现成已有的APL, PL/I软件(如编译程序和模拟程序等),而且这些软件是很多机器都已配上了的。

软件和硬件都是对信息流进行处理,两者是有很多相似、对应之处的。对于软件,是用顺序对一定的数据结构进行加工;对于硬件,是用控制部分对数据通路进行操作。例如,可以把寄存器看作向量,存储器看作二维数组;而只读存储器可相当于常数,算术逻辑操作可相当于表达式,各种有条件的硬件操作可相当于条件语句等等。因此,程序设计语言中的很多描述符是可以直接引用的。但是,高级程序设计语言之所以“高级”,就是它应尽量与具体机器的组成特点无关,它们都是朝着远离具体机器结构的方向发展;现在反过来,希望用它来描述具体机器硬件就比较为难了。譬如,象机器的硬件组成、复杂的操作时间关系、操作速度(传输延迟)、并行操作等特性,直接引用高级程序设计语言是难以描述的。突出的是不适于描述硬件中的并行性,而并行操作正是硬件设计的主要目标之一。因此,有人对这些高级语言作某些修改使其更适于描述硬件,1971年提出的AHPL(A Hardware Programming Language)硬件描述语言就是基于APL,按硬件描述要求作了修改,适于描述复杂模块的语言。另一种途径是新设计专用于硬件描述的语言。至今,提出了很多种这样的语言,其中,1965年美国朱耀汉教授提出的CDL(Computer Design Language)语言适合于门及寄存器的描述,在欧洲有广泛的应用;1967年提出的DDL(Digital Design Language)语言则适于描述大部件中寄存器间的数据传送;1971年提出的ISP(Instruction Set Processor)语言适于描述设计要求,而PMS(Process Memory Switch)语言则适于描述硬件的组成。这种途径的缺点主要是要自己设计模拟程序,加之这些语言本质上差别不大,但又统一不起来,因而没能得到普遍应用,每种硬件描述语言都只是在一种或个别几种型号机器上配置了它的模拟程序。不过,每个大的计算机厂家内实际上都已有了它们自己用的主要用于模拟硬件的硬件描述语言,但很少公布于众。

七十年代,进一步重视对硬件描述语言的研究,除研究用于描述和模拟的语言,还研究用于设计的语言,当然这几方面是密切相关的。用于模拟的语言,要求用它能编写出精确描述硬件动作,并能对其正确性进行校验的程序。用于设计的语言,还要求能说明设计要求并能用它进行结构式设计;联系到分布处理与并行处理以及LSI、VLSI和微处理器的发展,一个计算机系统内可能要用到多种强功能的大规模组件和多种型号的微处理器,因此,需要有不通过具体的门、寄存器逻辑图就能准确描述这些片子的内部功能及引出腿的文本式语言,才能进行自动设计,这种语言对于手工设计同样也有好处。

到了七十年代末,要求有统一的硬件描述语言的呼声日高。就是从设计文件的编制、产品说明书的编写以至便于教科书和文献的阅读,都希望有统一的文本式及图形式硬件描述语言。当然,统一语言的需要远不只是这些,它有着更为深远的意义。例如,一种型号的大规模集成电路片子,可能被多个整机厂选用,用于构成不同型号的机器,只有有了统一的硬件描述语言,器件厂才能为其生产的片子提供文本式语言描述的功能说明,各个整机厂也才能方便、准确地使用它。又如,有了统一的描述语言,就为各种LSI片子,以至各种机器的性能比较,提供很大的方便。

研究硬件描述语言的意义不只是促使更多的硬件设计者探索如何用“软”的办法进行设计,即用计算机辅助设计计算机;而且还进一步促使了软、硬的结合。随着硬件价格的持续下降,软件造价的不断上升,在计算机系统内用便宜、可靠的硬件替代贵的、不可靠的软件(前已说过,软件的不可靠性指的是难于设计出完全正确无误的软件),这个趋势在八十年代将更为明显。为实现这点,就需要设计出既能描述软件结构又能描述硬件结构的语言。因为,从本质上看,软件和硬件都是实现某种算法,只是手段和方法不同。因此,从更高的角度看,研究出既能描述硬件,又能描述软件的设计语言是完全可能的。

下面介绍二种硬件描述语言:文本式的CDL语言和一种交互式的图形语言。这里,不打算详述各种语言的全部细节,主要是通过几个例子说明每种语言的特点,使之对硬件描述语言能有一个概念。选取这二种语言,并不是因为它们已是普遍使用的公认硬件描述语言。

1.2 计算机设计语言 CDL

1964年在美国马里兰大学研制出一种硬件描述语言CDL。1965年由该大学朱耀汉教授第一次发表,并接着提供能用于IBM 7094的模拟程序。

1.2-1 CDL 描述符

CDL语言能直接描述寄存器、部分寄存器、存储器、译码器、开关逻辑、时钟和一组并行的微操作等等。下面给出这些描述符和一些约定,以及常用的一些语句格式。

一、常量和变量

在CDL中约定,所用的常量都是十进制正整数,如果要用其它进制数需另外标注。变量是由一个或多个字母或字母和数字组成的字符串,而且打头的一个字符必须是字母。要求变量名字必须是唯一的。对组成计算机的元、部件,如: Register, Subregister, Memories, Decoders, Switches, Terminals, Clocks 和 Blocks 等都当作变量来处理。所有变量名称必须事先在某个说明语句中给予说明后,才可以引用。

二、说明语句

CDL 的说明语句格式列于表 8.1。这些说明语句类似于程序设计语言中的数据类型说明语句。

(1) 寄存器说明语句: REGISTER, list

其中 list 是指一组寄存器的名单。每个定义符是由变量名字跟一对括号构成。括号内有用短线隔开的两个十进制数, 指明该寄存器中的各个位从左到右的编址号。若只有一位, 则括号可省去。

表 8.1 说明语句

说明语句	格 式	例
寄存器	REGISTER, list,	REGISTER, A(0-5), B2(10-1), C
部分寄存器	SUBREGISTER, list	SUBREGISTER, B2(OP) = B(4-1), A(OR) = A(0-3)
存贮器	MEMORY, list	MEMORY, M(\$)=M(0-63, 0-10), N(J)=N(0-6, 3-1)
译码器	DECODER, list	DECODER, K(0-1)=F, L(0-15)=G(2-5)
时 钟	CLOCK, P(d)	CLOCK, P(2)
开 关	SWITCH, list	SWITCH, START(OFF, ON), SENSE(P1, P2, P3)
组合逻辑	TERMINAL, list	TERMINAL, T(1-4)←A*B+C, R(6-1)←G(2-7)*1
微语句块	BLOCK, list	BLOCK, PAR(A←B, R←0), CYC(A←Acountup)

例: REGISTER, A(0-5), B2(10-1), C

该语句定义了三个寄存器: A 寄存器有 6 位, 从左到右编号为 0, 1, 2, 3, 4, 5; B2 寄存器有 10 位, 从左到右编号为 10, 9, 8, ..., 1; 而 C 寄存器只有 1 位。

(2) 部分寄存器说明: SUBREGISTER, list

例: SUBREGISTER, B2(OP) = B2(4-1), A(OR) = A(0-3)

部分寄存器名称总是要带有一个寄存器名字, 表明它是该寄存器的一部分。部分寄存器名单是一组用逗号隔开的等式。在出现这个部分寄存器说明语句之前, 所有等式右边用的变量都必须是已说明过的。譬如, 例中的 A, B2 等应是已说明过的寄存器。

(3) 存贮器说明: MEMORY, list

例: MEMORY, M(\$)=M(0-63, 0-8), N(J)=N(0-6, 3-1)

名单中定义的每个存贮器有如下格式:

$$M(R) = M(d_1 - d_2, e_1 - e_2)$$

其中 M 是存贮器名字, R 是其地址寄存器的名字。R 必须是事先已说明过的寄存器。 d_1 和 d_2 是两个十进制数, 指明存贮器 M 的字编址范围, e_1 和 e_2 也是两个十进制数, 表示每个字的位编址范围。在存贮器说明语句之后, 就可以用常数地址调用该存贮器中的某一个字, 譬如用 M(12) 调用地址编号为 12 中的字, 或者用它的地址寄存器指定一个字, 如 M(R), 指明 R 中的内容就是被调用字的地址。也可以访问存贮器中一个字的某几位, 例如: M(R)(0-2, 4, 4), 其中 0-2, 4, 4 就是要访问位的编号。

(4) 译码器说明: DECODER, list

例: DECODER, K(0-1)=F, L(0-15)=G(2-5)

名单中每个译码器定义符是由译码器名字后跟一对括号 (括号内有用短线分开的两个十进制数), 等号和一个以前已说明过的寄存器或寄存器的一部分所构成。

每个译码器定义符都给出与它相联的寄存器或寄存器的一段。在调用时要给出译码器名字跟以括号，括号内包含一个常数，例如， $L(5)$ 它的值是“1”或是“0”取决于在调用时刻与该译码器相联的寄存器内容是否等于括号内的常数。

(5) 时钟说明: $CLOCK, P(d)$

P 是时钟的名字， d 是十进制数，定义了 $d+1$ 个时钟脉冲，这些时钟脉冲在语句中可以这样调用: $P(0), P(1), \dots, P(d)$ 。这些时钟的时间关系如图 8.3 所示。

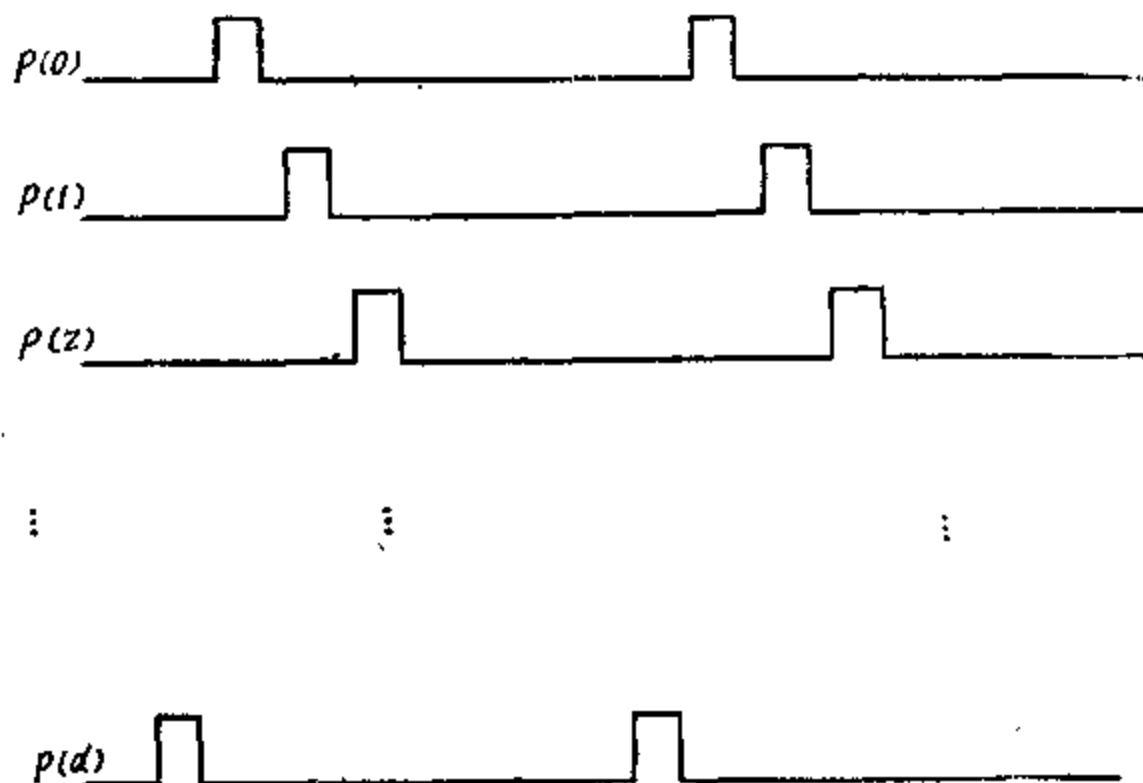


图 8.3 多相位时钟定时图

例: $CLOCK, P(2)$

(6) 开关说明: $SWITCH, list$

例: $SWITCH, START(OFF, ON), SENSE(P1, P2, P3)$

每个开关定义符是由开关名字跟以一对括号（括号内是用逗号隔开的开关位置）构成。括号内第一个位置是开关在开始时处的位置，也就是说该开关在模拟时的初始状态。在所举的例子中，当模拟开始时，开关 $START$ 处于 OFF 的位置，开关 $SENSE$ 处于 $P1$ 位置。对一个开关的调用，不是查看一下它所处的位置，就是给它置于某一位置。查看开关位置，可以用这样的格式: $NAME(POS)$

例: $SENSE(P2)$ 它给出“是”或“非”值取决于调用时该开关所处的位置。欲将开关置于某一位置可以用下面的格式: $NAME = POS$ 。

例: $SENSE = P2$

(7) 组合逻辑网络说明: $TERMINAL, list$

例: $TERMINAL, T(1-4) \leftarrow A \cdot B + C, R(6-1) \leftarrow G(2-7) \cdot 1$

每个逻辑网络用一个微语句定义，即用一个网络名字跟一个替代符(\leftarrow)和一个表达式构成。出现在表达式中的变量必须事先说明过。如果表达式是一个多位置量，则必须在网络名字后的括号内给出编址约定。

(8) 微语句块说明: $BLOCK, list$

例: $BLOCK, PAR(A \leftarrow B, R \leftarrow 0), CYC(A \leftarrow A \text{ countup})$

每个块由块名字跟以括号构成，括号内是用逗号隔开的一串微语句。在所举的例子中，定义了两个微语句块：PAR 和 CYC。PAR 块中包含两个微语句：用 B 替代 A，给 R 置“0”，而 CYC 块只有一个微语句：将 A 内容加 1 再放入 A 中。

三、基本操作符和表达式

(1) 基本操作符

CDL 基本操作符列于表 8.2。另外，还可以根据基本操作作用独立的子程序定义特殊操作符。

表 8.2 基本操作符

操 作 符	定 义
,	取 补
←	替 代
-	联 接
+	逻 辑 或
*	逻 辑 与
=	相 等
≠	不 等

(2) 表达式

表达式是由常数、变量、操作符（替代符除外）和具有通常数学意义的括号等组成。

例如， $(B - B) * C + 1, D, 10$

都是正确的表达式。

当表达式中没有用括号明确规定操作层次时，约定各种操作的优先次序为：取补，相等、不等、特殊操作，逻辑与，逻辑或，联接。如果有括号，则先括号内，后括号外；同一括号内取补最先做，联接最后做。

四、微语句

(1) 无条件微语句

无条件微语句是由用存贮单元表示的变量、替代符和一个表达式组成。

例： $A \leftarrow 1, B(1, 3 - 5) \leftarrow C * D + E(2, 0 - 2)$

都是正确的微语句。替代符左、右定义的位数必须相等；左边的某一位不可调用两次以上。

(2) 条件微语句

格式：IF (表达式) THEN (微语句)

或

IF (表达式) THEN (微语句) ELSE (微语句)

这里的表达式应给出值“1”或“0”，表示“是”或“非”。如果 IF 后面的表示式的值是“1”，在 THEN 后面括号内的所有用逗号隔开的微语句都要执行；若该条件微语句中有 ELSE，ELSE 后面括号内所有微语句都跳过去不执行。相反的，若 IF 后面表达式的值是“0”，THEN 后面的括号内的微语句都不执行，而 ELSE 后面括号内的微语句都要执行。

五、开关语句

格式: /NAME(POSITION)/ 微语句

其中 NAME 是一个已说明过的开关名称, POSITION 是该开关的某一个位置。当此开关置于括号内所指的位置时,后面的微语句都被执行。欲把开关置于某一状态(位置),可以手动,也可以用一个微语句: NAME = POS

例: SWITCH, START(OFF, ON), SENSE(S1, S2, S3)

/START(ON)/ SENSE = S2

若开关 START 在 ON 位置,例中的开关语句就被执行,结果,开关 SENSE 被置于 S2 的位置。

六、标号语句

格式: /Label/ 微语句

其中 Label 是标号,它应该如下构成:

Label = 表达式 * 时钟

Label 中的表达式不能与任何时钟有关系,而且其值必须给定是“1”或“0”。当 Label 的给定值为“1”时,后面跟着的微语句全部被执行;若为“0”,则都不执行。

标号语句中的微语句描述了各个元、部件之间的联系。计算机中与时间有关的操作都用标号语句描述。注意,标号语句中的微语句都是并行的,并假定替代操作的延迟可以忽略。

例如 /K(O)*P/ A \leftrightarrow B, B \leftrightarrow A

认为该语句实现的是 A 与 B 的内容交换。

特殊的情况,标号可能没有时钟,只有表达式。在这种情况下,标号后面的微语句中,必须包含一个微语句,将标号值变成“非”。例如

/A(5)/ ..., A(5) \leftarrow 0, ...

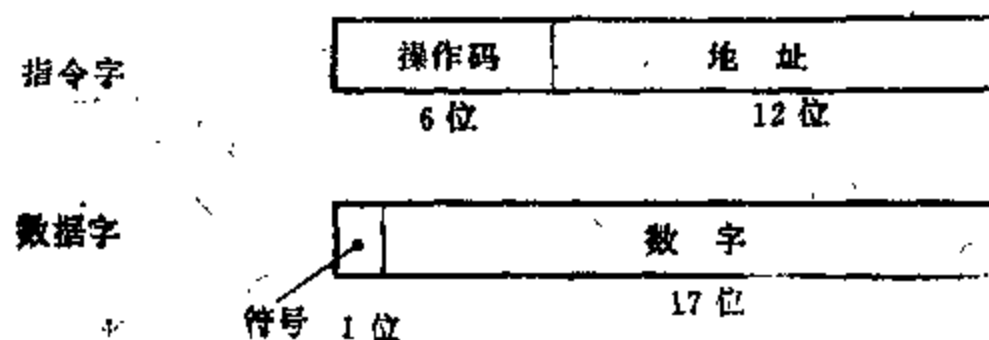
其它,如结束语句、注释语句和标头语句等等与程序设计语言中的相似,就不再赘述。

1.2-2 用 CDL 描述一台计算机

下面用 CDL 描述一台很简单计算机的结构组成及指令操作流程。目的是使大家对使用 CDL 有些具体的了解,先给出其指令系统、字格式、组成结构和操作序列流程图。

(1) 指令系统与字格式

本机字长 18 位,有指令字和数据字二种格式如下所示:



本机有九种指令: ADD, SUB, IOM, STO, JMP, SHR, CIL, CLA 和 STP。示于表 8.3。

其中, m 为操作数地址, n 为指令地址。这里有三条无地址指令,它们不访问存储器,只对

表 8.3 操作码与符号指令编码

符 号	名 称	符 号 编 码	操 作 码
ADD	加	ADD m	0 0
SUB	减	SUB m	0 1
JOM	按负转移	JOM n	0 2
STO	存	STO m	0 3
JMP	转移	JMP n	0 4
SHR	右移	SHR	0 5
CIL	循环左移	CIL	0 6
CLA	清加	CLA m	0 7
STP	停机	STP	1 0

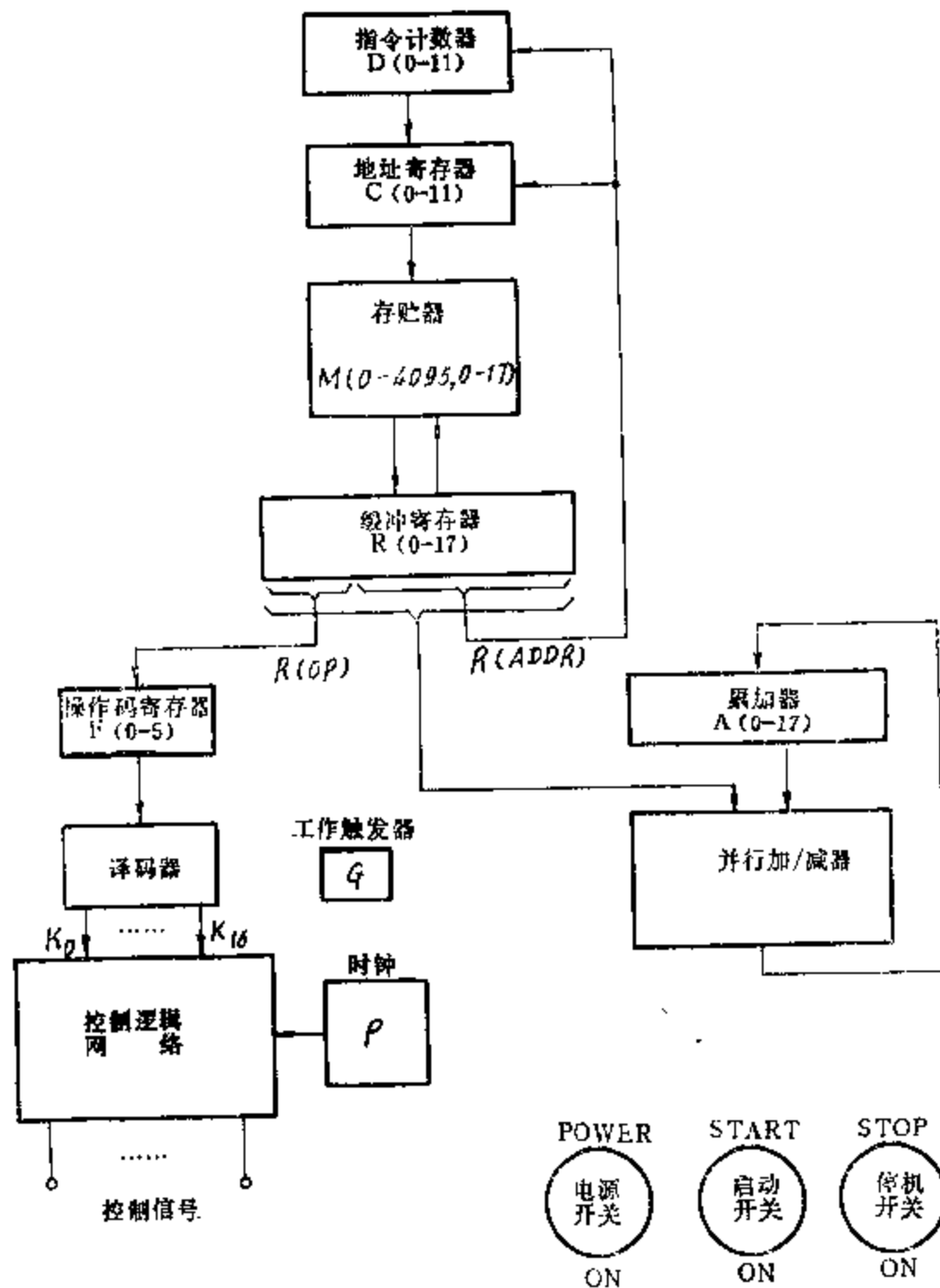


图 8.4 计算机组成结构

累加器中的数进行操作或是停机。操作码用八进制表示。

(2) 结构组成

该计算机有一个随机访问存储器，六个寄存器，三个开关，一个译码器，一个并减器和一个时钟。其结构图示于图 8.4。D 寄存器存的是下一条指令的地址。操作码寄存器与时钟配合产生十种操作序列的控制命令。工作触发器 G 置“1”时计算机处于工作状态，反之，处于等待状态。移位操作在 A 进行。加、减运算时把数当作无符号数处理，有考虑溢出指示。

(3) 操作序列流程

所有指令的操作序列流程示于图 8.5。分为启动序列、等待序列和执行序列。当合上电源开关时，启动工作触发器 G 被置于“0”；C、D 寄存器一直被置“0”，计算机处于等待状态，见图中的等待循环。若合上启动开关 START，启动触发器 G 被置为“1”，计算机走出等待循环进入工作状态。因为这时 C 寄存器为“0”，所以是从“0”号地址开始执行程序。

下面举例说明取指令序列与执行指令序列的微操作。

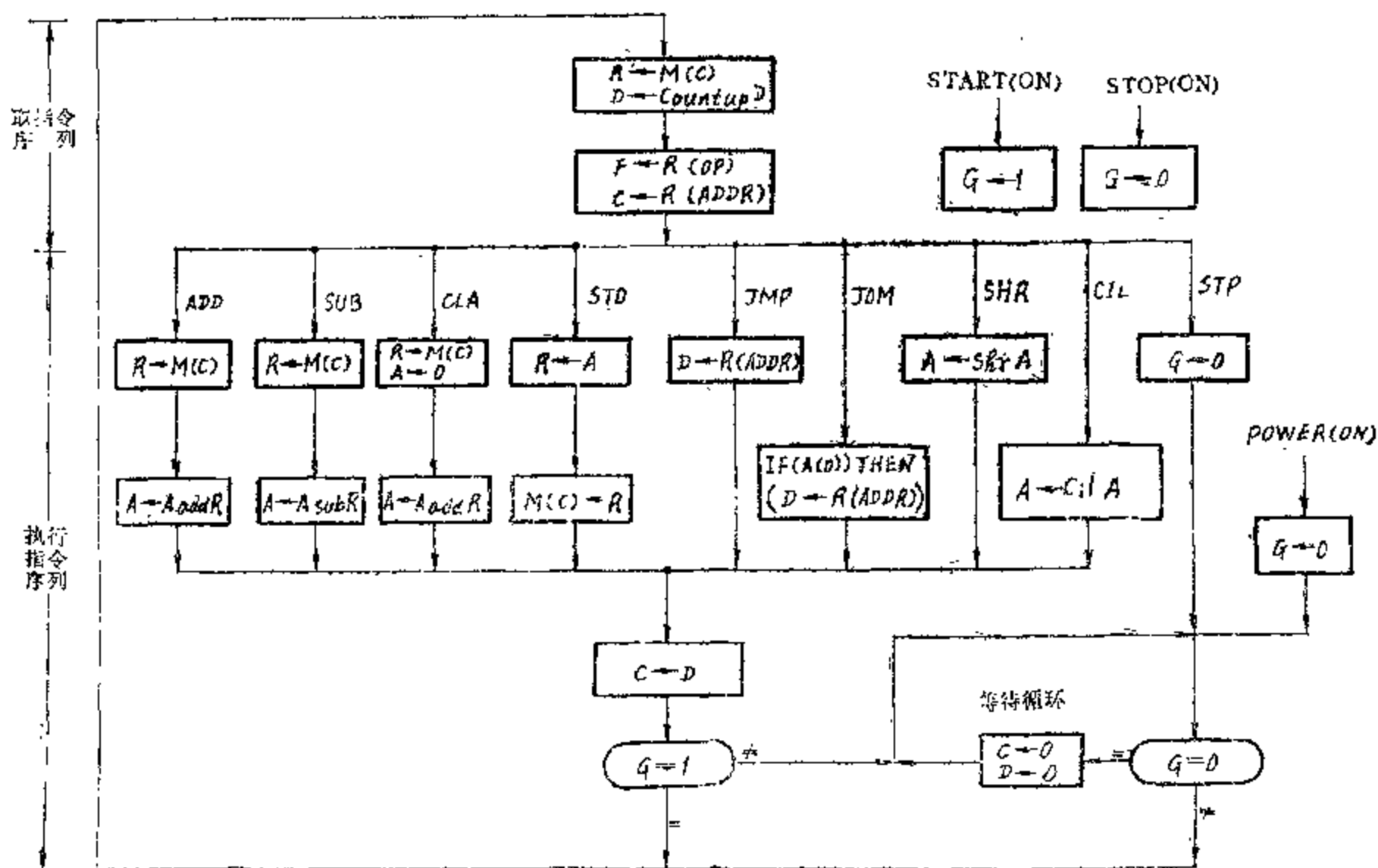


图 8.5 操作流程

取指令序列包括以下四个微操作，分两步完成，

1. $R \leftarrow M(C)$ $D \leftarrow \text{countup } D$
2. $F \leftarrow R(OP)$ $C \leftarrow R(ADDR)$

第 1 步中，头一个微操作将指令取至 R 寄存器。第 2 步中的两个微操作分别把指令操作码送至控制寄存器 F，地址码送至地址寄存器 C。

执行指令序列随不同的操作码而异。若操作码是00,就开始加法序列,执行以下微操作:

1. $R \leftarrow M(C)$
2. $A \leftarrow A \text{ add } R$
3. $C \leftarrow D$

第一步从内存M取操作数放入R,第二步将R与累加器A的内容相加,第三步是将下条指令的地址送往地址寄存器C。在完成时,如 $G=1$,取下一条指令继续工作下去;若 $G=0$,就停机。

若操作码是04,就开始转移指令JMP的执行序列。该序列分两步:

1. $D \leftarrow R(ADDR)$
2. $C \leftarrow D$

第一个微操作将R中的地址码部分(转向去址)送往程序计数器D。

如果操作码是02,就开始按负转移JOM的执行序列,其微操作如下:

1. IF(A(0)) THEN($D \leftarrow R(ADDR)$)
2. $C \leftarrow D$

对应于九条指令的执行序列都很简单,就不再列举了。

(4) CDL 描述文本

REGISTER,	R(0-17),	\$ 存储器的缓冲寄存器
	A(0-17),	\$ 累加器
	C(0-11),	\$ 存储器的地址寄存器
	D(0-11),	\$ 程序计数器
	F(0-5),	\$ 控制寄存器
	G	\$ 工作控制触发器
SUBREGISTER,	R(OP) = R(0-5),	\$ R寄存器的操作码部分
	R(ADDR) = R(6-17)	\$ R寄存器的地址码部分
MEMORY,	M(0-4095, 0-17)	
DECODER,	K(0-16) = F	\$ 控制命令发生器
SWITCH,	POWER(ON),	\$ 电源开关
	START(ON),	\$ 启动开关
	STOP(ON)	\$ 停机开关
CLOCK,	P	\$ 单拍时钟
TERMINAL,	ADD = K(0),	\$ 加命令
	SUB = K(1),	\$ 减命令
	JOM = K(2),	\$ 按负转移命令
	STO = K(3),	\$ 存贮命令
	JMP = K(4),	\$ 跳转命令
	SHR = K(5),	\$ 右移命令
	CIL = K(6),	\$ 循环左移命令
	CLA = K(7),	\$ 清加命令
	STP = K(8),	\$ 停机命令

FETCH = K(9),	\$ 取指令命令
WAIT = K(10)	\$ 等待命令

Comment, here begins the start-stop sequence.

/POWER(ON)/ G←0, F←12_s, C←0, D←0

/START(ON)/ G←1

/STOP(ON)/ G←0

Comment, here begins the wait sequence.

/WAIT*P/ IF(G=0) THEN(C←0, D←0) ELSE(F←11_s).

Comment, here begins the fetch sequence.

/FETCH*P/ R←M(C), D←countup D, F←15_s

/K(13)*P/ F←R(OP), C←R(ADDR)

Comment, here begin the execution sequences.

Comment, here begins the stop sequence.

/STP*P/ G←0, F←12_s

Comment, here begins the add sequence.

/ADD*P/ R←M(C), F←14_s

/K(12)*P/ A←A add R, F←13_s

Comment, here begins the sub sequence.

/SUB*P/ R←M(C), F←17_s

/K(15)*P/ A←A SUB R, F←13_s

Comment, here begins the jump sequence.

/JMP*P/ D←R(ADDR), F←13_s

Comment, here begins the jump on minus sequence.

/JOM*P/ IF (A(0)) THEN (D←R(ADDR)), F←13_s

Comment, here begins the store sequence.

/STO*P/ R←A, F←20_s

/K(16)*P/ M(C)←R, F←13_s

Comment, here begins the shift-right one-bit sequence.

/SHR*P/ A←shr A, F←13_s

Comment, here begins the circulate-left one-bit sequence.

/CIL*P/ A←cil A, F←13_s

Comment, here begins the clear add sequence.

/CLA*P/ R←M(C), A←0, F←16_s

/K(14)*P/ A←A add R, F←13_s

/K(11)*P/ C←D, IF(G) THEN (F←11_s) ELSE(F←12_s)

END

图 8.6 用 CDL 语言描述的计算机

在这个 CDL 描述文本里, 开头用说明语句描述其结构组成, 接着用标号语句描述了对这些组成部件的操作及其时间关系, 中间插入了注释语句。这里描述了九条指令的操作流

程。除了停机指令外，每条指令执行的末尾都要将操作码寄存器 F 置为 11 ($F \leftarrow 13_2$)，即最后一条标号语句：/K(11)*P/ $C \leftarrow D$, IF.....，是每条指令序列都要执行的。

看出，用 CDL 语言描述计算机的基本构成以及指令流程是简明、清晰的。这台机器是硬联控制的，下面再以微程序控制机器为例来说明 CDL 的应用。

1.2-3 用 CDL 描述一台微程序计算机

(1) 结构组成

这台计算机的结构组成如图 8.7 所示。D, C, R, A 寄存器的作用如上述例子，此外，还有只读控存 CM 及其地址寄存器 H 和缓冲寄存器 F。

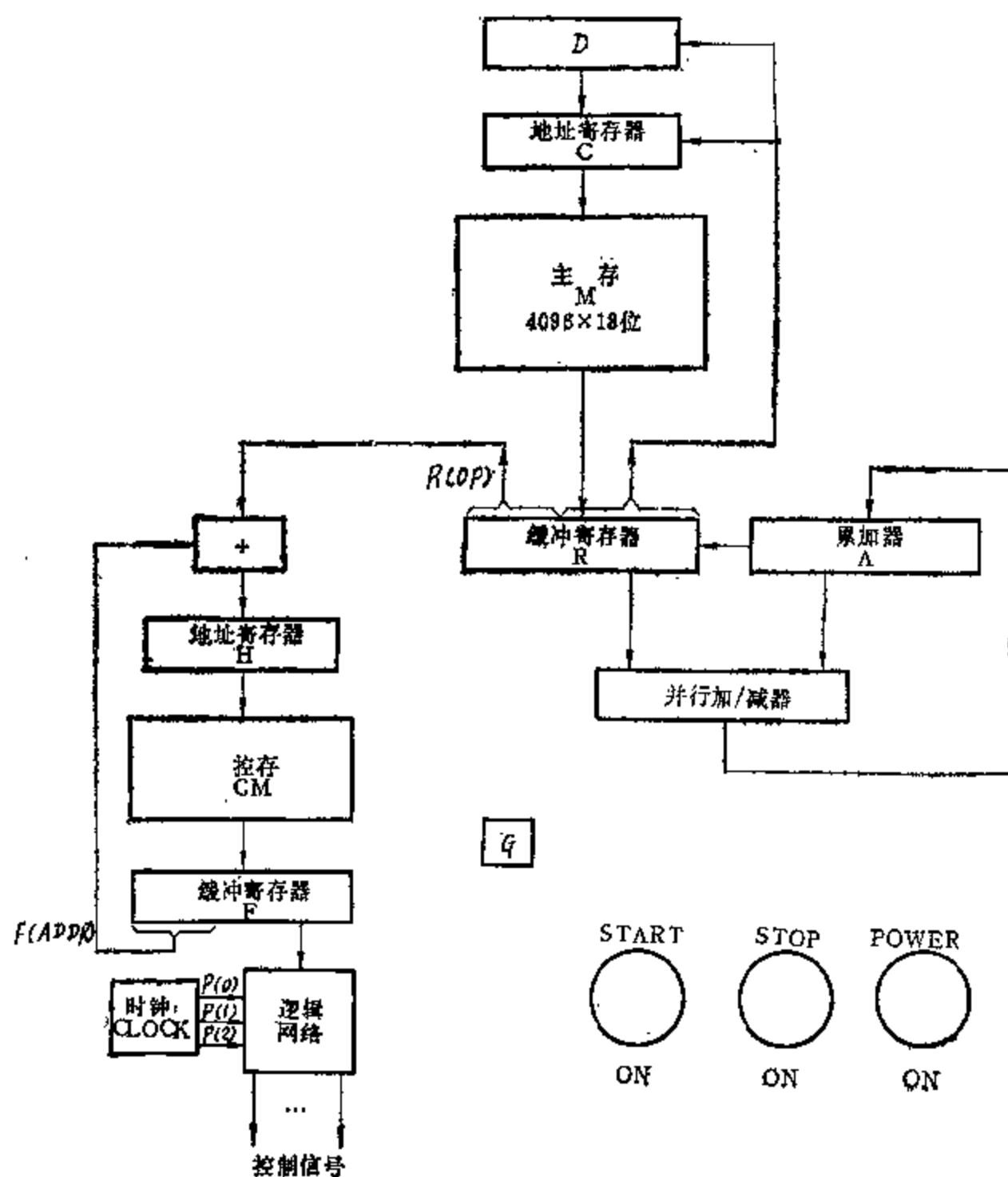


图 8.7 一台微程序控制计算机的结构

(2) 控制信号

一个主存读/写周期等于一个时钟周期，分三拍执行。在上一个时钟周期最末一拍 P(2) 将地址送往地址寄存器 C。如果是读周期，在下一个时钟周期的第一拍 P(0) 将读出的内容送至 R 寄存器；如果是写周期，则是在 P(1) 才把 R 中的数写入存贮器。这样的读/写主存周期用描述符描述如下。

读周期是:

/P(2)/ $C \leftarrow R(ADRS)$ or $C \leftarrow D$ \$一个主存周期的开始
 /P(0)/ $R \leftarrow M(C)$
 /P(1)/ \$一个主存周期的结束

写周期是:

/P(2)/ $C \leftarrow R(ADRS)$ or $C \leftarrow D$ \$一个主存周期的开始
 /P(0)/
 /P(1)/ $M(C) \leftarrow R$ \$一个主存周期的结束

一个控存周期分两拍完成:

/P(1)/ $H \leftarrow R(OP)$ or $H \leftarrow F(ADDR)$ \$一个控存周期开始
 /P(2)/ $F \leftarrow CM(H)$ \$一个控存周期结束

$R(OP)$ 为指令操作码, $F(ADDR)$ 为下字址。

(3) 微指令格式

微指令有 18 位长, 前 5 位, 即 $F(1-5)$, 是下字址字段, 其余为控制位。 $F(6-18)$ 与时钟节拍组合产生了各种微操作控制信号, 列于表 8.4。

表 8.4 这台微程序计算机的微指令格式及其微操作

控制位	时钟节拍	微 操 作
$F(1-5)$		控 存 下 字 址 字 段
$F(6)$	$P(0)$	$R \leftarrow M(C)$
$F(7)$	$P(1)$	$H \leftarrow R(OP), C \leftarrow R(ADRS), D \leftarrow \text{countup } D$
$F(8)$	$P(2)$	$IF(G) THEN (F \leftarrow CM(H)) ELSE (H \leftarrow 0, C \leftarrow 0, D \leftarrow 0, R \leftarrow 0)$
$F(9)$	$P(1)$	$H \leftarrow F(ADDR)$
	$P(2)$	$C \leftarrow D, F \leftarrow CM(H)$
$F(10)$	$P(0)$	$R \leftarrow A$
	$P(1)$	$M(C) \leftarrow R$
$F(11)$	$P(1)$	$A \leftarrow A \text{ add } R$
$F(12)$	$P(1)$	$A \leftarrow A \text{ sub } R$
$F(13)$	$P(0)$	$D \leftarrow R(ADRS)$
$F(14)$	$P(0)$	$IF(A(0)) THEN (D \leftarrow R(ADRS))$
$F(15)$	$P(0)$	$A \leftarrow shr A$
$F(16)$	$P(0)$	$A \leftarrow cil A$
$F(17)$	$P(0)$	$A \leftarrow 0$
$F(18)$	$P(0)$	$G \leftarrow 0$

(4) CDL 描述文本

REGISTER, R(0-17),	\$ 存储器M的缓冲寄存器
A(0-17),	\$ 累加器
C(0-11),	\$ 存储器M的地址寄存器
D(0-11),	\$ 程序计数器
F(0-18),	\$ 控存CM的缓冲寄存器
H(1-5),	\$ 控存CM的地址寄存器
G	\$ 启停控制触发器
SUBREGISTER, R(OP) = R(1-5),	\$ R寄存器的操作码部分
R(ADRS) = R(6-17),	\$ R寄存器的地址码部分
F(ADDR) = F(1-5)	\$ F寄存器的地址码部分
MEMORY, M(C) = M(0-4095, 0-17),	\$ 主存储器
CM(H) = CM(0-11, 1-18),	\$ 控制存储器
SWITCH, POWER(ON),	\$ 电源开关
START(ON),	\$ 启动开关
STOP(ON)	\$ 停机开关
CLOCK, P(0-2)	\$ 三相时钟

Comment, start computer operation

```

/POWER(ON)/ G←0, F←0, H←0, C←0, D←0
/START(ON)/ G←1, F(8)←1
/STOP(ON)/ G←0

```

Comment, fetch sequence when H = 0

```

/F(6)*P(0)/ R←M(C)
/F(7)*P(1)/ H←R(OP), C←R(ADRS), D←countup D
/F(8)*P(2)/ IF(G) THEN(F←CM(H)) ELSE(H←0, C←0, D←0, R←0)

```

Comment, ADD sequence when H = 1 (op-code = 01)

```

C/F(6)*P(0)/ R←M(c)
/F(11)*P(1)/ A←A add R
/F(9)*P(1)/ H←F(ADDR)
/F(9)*P(2)/ C←D, F←CM(H)

```

Comment, SUB sequence when H = 2 (op-code = 02)

```

C/F(6)*P(0)/ R←M(C)
/F(12)*P(1)/ A←A sub R
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)

```

comment, JOM sequence when H = 3 (op-code = 03)

```

/F(14)*P(0)/ IF(A(0)) THEN (D←R(ADRS))
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)

```

```

Comment, STO sequence when H=4 (op-code=04)
  /F(10)*P(0)/ R←A
  /F(10)*P(1)/ M(c)*R
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)
Comment, JMP Sequence when H=5 (op-code=05)
  /F(13)*P(0)/ D←R(ADRS)
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)
Comment, SHR sequence when H=6 (op-code=06)
  /F(15)*P(0)/ A←Shr A
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)
Comment, CIL sequence when H=7 (op-code=07)
  /F(16)*P(0)/ A←cil A
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)
Comment, CLA sequence when H=8 (op-code=10)
C/F(6)*P(0)/ R←M(C)
  /F(17)*P(0)/ A←0
C/F(11)*P(1)/ A←A add R
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)
Comment, STP sequence when H=9 (op-code=11)
  /F(18)*P(0)/ G←0
C/F(9)*P(1)/ H←F(ADDR)
C/F(9)*P(2)/ C←D, F←CM(H)
  END

```

图 8.8 微程序计算机的 CDL 描述文本

在图 8.8 的 CDL 文本中, 开头用说明语句给出该计算机的结构组成, 接着用标号语句描述该计算机的指令操作及控制流程, 共分 11 组, 每组由注释语句开始。第一组描述一组手动操作; 第二组是取指令序列; 其余 9 组, 每组对应一条指令执行序列。在文本中, 前头注有字母 C 的标号语句是多余的, 实际上被当作注释语句处理。因为 CDL 是非顺序性语言, 只要语句标号的控制条件满足, 该语句就被执行, 与写在文本中的位置无关, 所以重复写相同的语句是多余的。这里, 我们这样写只是为了增加可读性。如果去掉这些多余的语句, 描述一台微程序计算机比上一个例子中描述的硬联计算机还要简炼。

在这里, 我们不必给出流程图, 只要熟悉 CDL 语言的语义, 不用流程图而从描述文本也能很快看出其动作流程。参看图 8.8, 合上电源开关, POWER(ON)条件满足, 所有寄存器

都被置“0”。若再合上启动开关，START(ON)条件满足，启停控制触发器G置“1”，控存中F寄存器第8个控制位被置“1”。时钟节拍循环至P(2)时，/F(8)*P(2)/标号条件满足，其后面的条件微语句：

IF(G) THEN(F←CM(H)) ELSE(H←0, C←0, D←0, R←0)

被执行。这时是执行(F←CM(0))，则控制地址为0的微指令被取出送至F寄存器。这条微指令的码点见表8.5的第一行，它控制取指令的操作。此时F寄存器中的6、7、8位都是“1”，从而使取指令序列的三个语句都能执行。这样，就从主存中地址为(C)（第一次取指令时C=0）的单元取出一条指令，并把操作码送至控存地址寄存器H，而地址码部分（可能是一个操作数的地址）送至主存地址寄存器C，准备下一次访问主存。在取指令序列最末一拍，已将与操作码对应的微指令取到F。

之后，由该操作码决定的操作序列开始执行。若从主存取出的是一条加法指令（其操作码为01₆），控存从H=01的地址取出控制加法指令的微指令放在F。其F(6)，F(9)，F(11)均为“1”，从而使得加法指令操作序列的所有微操作都被执行。在执行加法序列的末

表8.5 控存微指令码点

H	OP-code	F(1-5)	F(6)	F(7)	F(8)	F(9)	F(10)	F(11)	F(12)	F(13)	F(14)	F(15)	F(16)	F(17)	F(18)
0	FETCH	0	1	1	1	0	0	0	0	0	0	0	0	0	0
1	ADD	0	1	0	0	1	0	1	0	0	0	0	0	0	0
2	SUB	0	1	0	0	1	0	0	1	0	0	0	0	0	0
3	JOM	0	0	0	0	1	0	0	0	0	1	0	0	0	0
4	STO	0	0	0	0	1	1	0	0	0	0	0	0	0	0
5	JMP	0	0	0	0	1	0	0	0	1	0	0	0	0	0
6	SHR	0	0	0	0	1	0	0	0	0	0	1	0	0	0
7	CIL	0	0	0	0	1	0	0	0	0	0	0	1	0	0
8	CLA	0	1	0	0	1	0	1	0	0	0	0	0	1	0
9	STP	0	0	0	0	1	0	0	0	0	0	0	0	0	1

尾，F中又已存放好一条控制取指令的微指令，而这时主存地址寄存器C中已是下条指令的地址。所以，若不是执行停机指令，也不合上停机开关，就可以按主存中的程序一条条继续执行下去。若合上了停机开关或遇到一条停机指令，启停控制触发器G被置“0”。这样，在取指令序列的最后一拍，将所有寄存器置“0”，而后停止操作。可见，从CDL描述的文本看懂一台计算机的操作流程比从逻辑图看要简单。

至此，我们讲的是如何用CDL语言描述已有的数字系统，下面讲述，它在设计数字系统中的应用。

1.2-4 CDL语言在设计数字系统中的应用

本小节我们结合简单的例子，来说明在设计数字系统中如何使用CDL语言。

一个数字系统的设计大致可分为以下八个步骤：

- (1) 描述设计要求；
- (2) 选定寄存器结构；
- (3) 设计操作顺序；

- (4) 设计控制机构;
- (5) 用 CDL 给出设计;
- (6) 用 CDL 模拟器模拟设计;
- (7) 翻译成布尔方程;
- (8) 模拟逻辑设计。

下面举例说明前五个步骤。

(1) 描述设计要求

第一步用自然语言讲清楚需要硬件实现什么? 若现在要求的是将 5 位寄存器 A 的内容逐位串行取反。

(2) 选定寄存器结构

选定的寄存器结构如图 8.9 所示。

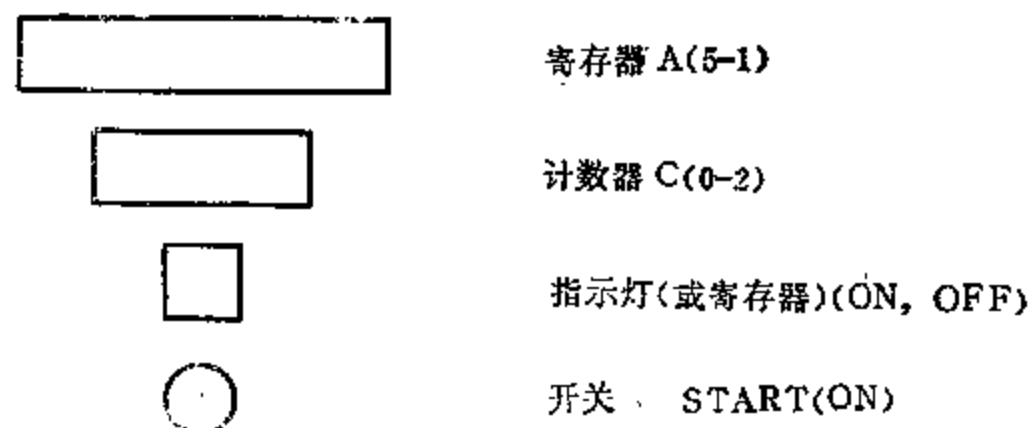


图 8.9 寄存器结构

用 CDL 描述如下:

```

REGISTER, A(5-1),    $ 移位寄存器
                  C(0-2)    $ 计数器
LIGHT, FINI(ON,OFF) $ 指示灯
SWITCH, START(ON)   $ 启动开关
  
```

(3) 确定操作顺序

完成上述设计要求的操作顺序如图 8.10 所示。它从接通开关 START 开始。在最上面的方框里, 用两个微操作将系统置为初始状态。在第二个方框里执行对 A(1)位求反和循环移位操作; 第二个微操作给计数器 C 加“1”。然后在下面的测试框里判计数器 C 内容是否已等于 5, 若 C=5, 进到下一框, 打开指示灯, 表示操作结束, 否则继续进行下一位的取反操作。

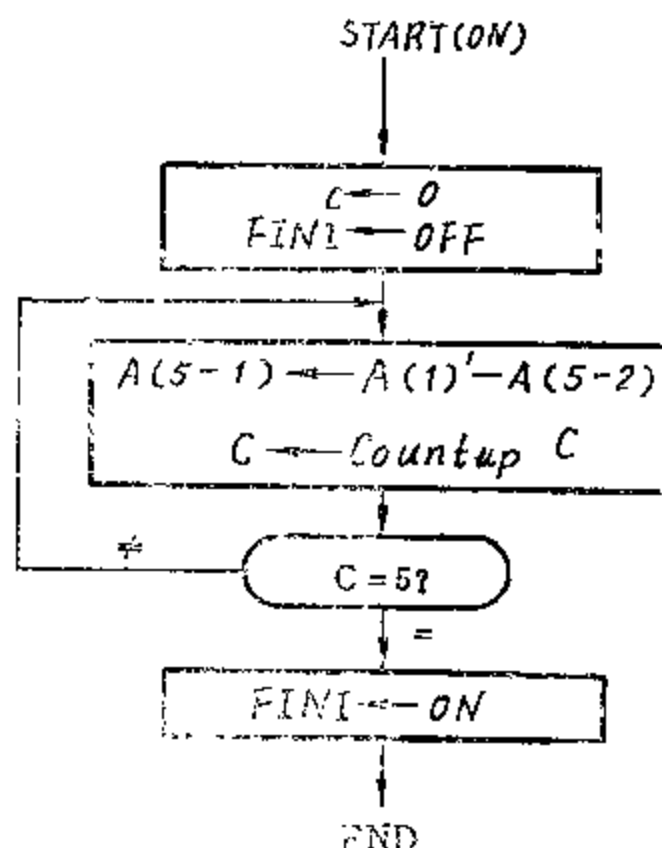


图 8.10 操作顺序图

(4) 设计控制机构

图 8.11 为所用的控制元件: 寄存器 T 和时钟 P。

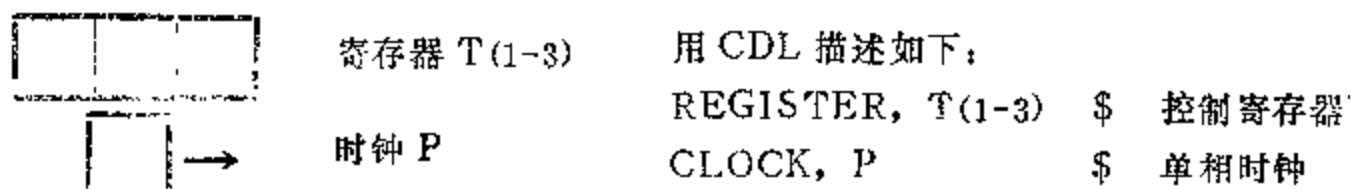


图 8.11 控制机构

先将图 8.10 的操作顺序图变成控制状态图，示于图 8.12。控制寄存器的三个状态分别用 $T(1)$ 、 $T(2)$ 和 $T(3)$ 来表示，循环次数用计数器 C 控制，以改变控制寄存器 T 的内容来改变状态。

用 CDL 描述如下：

```

/START(ON)/  T = 1002
/T(1)*P/      T = 0102
/T(2)*P/      IF(C = 5)
THEN (T ← 0012) ELSE (T ← 1002)
/T(3)*P/      T = 0
    
```

这里，每条执行语句都包括一个标号，并跟以一条或多条微语句。标号用一对斜线与微语句分开。标号表达式的二进制值为 1 时，跟着的微语句全部被执行。

在状态图中，若开关 $START$ 置于 ON 的位置，控制寄存器 T 被置成 100_2 ，在下一个时钟脉冲来时，将 T 变成 010_2 。接着的下一步进入什么状态，取决于计数器 C 的内容是否等于 5，若 $C = 5$ ， T 由 010_2 变为 001_2 ，若 $C \neq 5$ ， T 由 010_2 变为 100_2 。

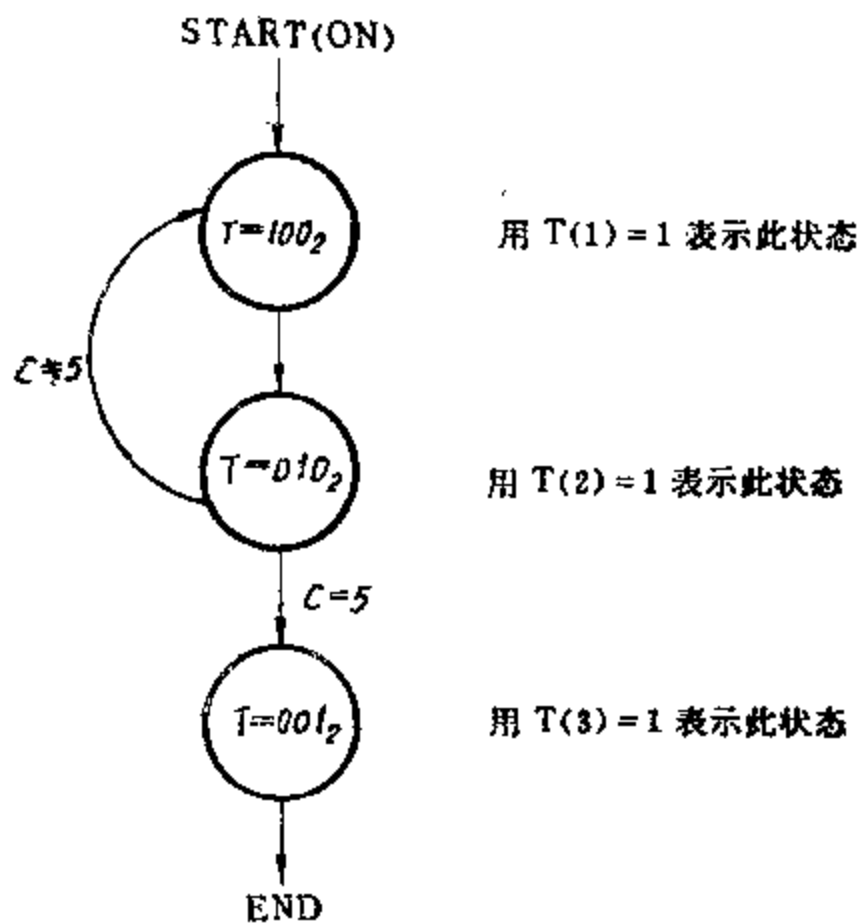


图 8.12 控制状态图

(5) 用 CDL 描述设计

用 CDL 描述的设计文本如图 8.13。

Comment, storager structure

```

REGISTER, A(5-1), C(0-2)
LIGHT,    FINI(ON, OFF)
SWITCH,   START(ON)
    
```

Comment, Processor structure

```

/START(ON)/ T ← 1002, C ← 0, FINI ← OFF
/T(1)*P/   T ← 0102, A ← A(1)' - A(5-2), C ← countup C
/T(2)*P/   IF(C = 5) THEN (T ← 0012) ELSE (T ← 1002)
/T(3)*P/   T ← 0, FINI ← ON
END
    
```

图 8.13 CDL 描述的设计文本

1.2-5 模拟测试

由以上例子可以看出，采用硬件描述语言的设计文本是简明扼要和规格化的。然而，采用硬件描述语言所带来的好处更在于能在构成硬件之前，就可对所做的设计进行模拟测试和正确性校验。所以，每种硬件描述语言都应有相应的模拟程序，CDL 也是如此。

CDL 的模拟程序实际上还包括翻译程序，翻译程序把经终端（或卡片机）来的设计文本翻译成适合于模拟用的格式（各种表和用伪指令写的程序）。模拟程序包括加载 (LOAD)，

输出 (OUTPUT)、开关 (SWITCH)、模拟 (SIMULATE)和恢复(RESET) 等五个例行程序。

加载例行程序控制接收数据包括测试程序、微程序码点并把它送到存贮器或指定的设计系统中的寄存器。输出例行程序处理模拟测试中所需要的输出,如在模拟过程中输出某些寄存器的内容、存贮器中某些单元的内容以及某些开关所处的位置等。开关例行程序模拟手动开关的操作。模拟例行程序则执行由翻译程序形成的、用伪指令写的程序。恢复例行程序则是在模拟结束时,将模拟程序本身恢复到初始状态。这五个例行程序的核心是模拟例行程序,它循环地执行以下几个步骤:

- a) 如果有开关手动操作出现,对应的开关语句中的微语句被执行;
- b) 计算所有的标号,标号值为“1”的在下一时钟为活跃标号;
- c) 活跃标号语句中的微语句,先是对存在各寄存器和某些存贮单元的值进行运算并收集运算结果,然后,将所收集的值存起来;
- d) 检查模拟过程是否结束,如果没结束再从步骤a)开始重复进行。

在模拟过程结束后,就可调用恢复例行程序。这时,如果还需要按另一组数据进行模拟,不必再调用翻译程序。

翻译程序可将翻译形成的各种表(包括子程序表、标号表、开关表、时钟表和变量符号表等)和用伪指令写的程序打印出来。翻译程序把所有的微语句、标号和逻辑网络的表达式以及译码器等都翻译成一系列伪指令程序。

模拟程序是用字符\$SIMULATE调用的。在调用了模拟程序之后就可以分别调用它所属的五个例行程序。

下面是模拟程序调用命令举例:

\$TRANSLATE	(调用翻译程序)
* MAIN	
.....	
.....	
END	
\$SIMULATE	(调用模拟程序)
* OUTPUT	
* SWITCH	
* LOAD	
.....	
.....	
* SIM.....	在模拟程序控制下
* RESET	的例行操作程序
* LOAD	
.....	
.....	
* SIM	

在这个模拟命令调用序列中,OUTPUT、SWITCH、LOAD程序的调用次序是任意的。

输出程序按输出名字表输出，它还能指明每隔多少标号（或时钟脉冲）输出一次。它给出的输出内容包括：标号周期数，时钟周期数，标号表达式为真值的标号表以及指定的寄存器和存贮单元的内容等。

只有在调用模拟例行程序之后，才真正开始模拟。在调用模拟例行程序时要给定停止模拟的条件。

例如：*SIM 400,3

就是指明模拟至时钟周期数为 400 时或每个时钟周期内接连的标号周期数超过 3 时停机。

恢复例行程序能恢复以前申请过的输出和以前要求过的手动开关操作，还能恢复时钟周期计数器和标号周期计数器。在调用恢复例行程序之后，就可以再调出输出、加载和模拟等例行程序。

模拟测试还能发现语法和语义上的错误，并能指明错误所在。CDL 可用多种信息格式指明错误。

例如：

```
ERROR IN MICROSTATEMENTS 7
ERROR IN LABELED STATEMENTS 3
TRANSLATOR ERROR 8
ERROR DURING SIMULATION 5
```

每种信息指明错误属哪一类，后面的数码则是指明错误是该类中的第几条错误。

例如：

```
ERROR IN MICROSTATEMENTS 7
```

指出错误属微语句错误中的第七条，即微语句的第一个符号或是非法的，或是还没有说明过的。又如，

```
TRANSLATOR ERROR 8
```

指明该语句是翻译程序识别不了的，而 ERROR DURING SIMULATION 5 则指明标号表达式不是单值的。

下面给出对前述硬联控制计算机和微程序控制计算机的 CDL 设计文本进行模拟测试的过程，并给出测试输出结果。

硬联控制计算机的模拟测试：

```
$TRANSLATE
* MAIN
.....
.....
END
$SIMULATE
* OUTPUT CLOCK(1) = R, A, C, D, F, G, M(100), M(101), M(102), M(103),
M(104)
* SWITCH 1, POWER = ON
* SWITCH 1, START = ON
* LOAD
```


M(0—) = 070100, 000101, 030102, 060000

M(4—) = 020006, 040003, 030103, 010102

M(10—) = 030104, 050000, 100000

M(100—) = 010001, 001001, 0, 0, 0

*SIM 400, 3

调用开关的语句: *SWITCH 1, POWER = ON

指明将开关 POWER 置向 ON 的手动操作发生在第一个标号周期之前。在调用加载程序 *LOAD 命令之后是加载的数据。例如, M(0—) =指明给存贮器 0 号单元开始的接连四个单元加载等号右边的四个数据。实际上, 这里加载的是表 8.6 的测试程序:

表 8.6 测试程序

存贮单元	单元内容	符号程序
0000	070100	CLA 0100
0001	000101	ADD 0101
0002	030102	STO 0102
0003	060000	CIL 0000
0004	020006	JOM 0006
0005	040003	JMP 0003
0006	030103	STO 0103
0007	010102	SUB 0102
0010	030104	STO 0104
0011	050000	SHR 0000
0012	100000	STP 0000
0100	010001	
0101	001001	
0102	011002*	
0103	440100*	
0104	427076*	

(注) 1. 带*号的数据是执行测试程序后得到的结果

2. 这里加载的数据都是八进制

在模拟过程中调用输出例行程序的语句是:

*OUTPUT CLOCK(1) = R, A, C,

它指明按每个时钟周期输出一次, 每次输出等号右边名单中的各寄存器和存贮单元的内容, 这可由图 8.14 给出的部分模拟输出结果看清楚。

头一段输出的是在第 1 个标号周期之前完成的开关操作状态及按调用输出例行程序的语句指定的名单顺序给出的各寄存器(R、A、C、D、F、G)和各存贮单元(000100, 000101, 000102, 000103, 000104)的内容。时钟周期开始计数后, 每隔一个时钟周期按名单顺序输出一一次各寄存器和各存贮单元的内容, 并同时给出标号周期计数以及为真值的标号。

实际上, 这里所说的模拟测试就是在一个宿主机上调用能识别 CDL 设计文本的模拟程序。先将被模拟机器的 CDL 设计文本的标号、微语句、表达式等翻译成各种表和一系列伪

SWITCH INTERRUPT

POWER = ON
START = ON

R = 000000 A = 400000 C = 0000 D = 0000 F = 12 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 1

TRUE LABELS
/WAIT * P/
CLOCK TIME = 1

R = 000000 A = 400000 C = 0000 D = 0000 F = 11 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 2

TRUE LABELS
/FETCH * P/
CLOCK TIME = 2

R = 070100 A = 400000 C = 0000 D = 0001 F = 15 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 3

TRUE LABELS
/K(15) * P/
CLOCK TIME = 3

R = 070100 A = 400000 C = 0100 D = 0001 F = 07 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 4

TRUE LABELS
/CLA * P/
CLOCK TIME = 4

R = 010001 A = 000000 C = 0100 D = 0001 F = 16 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 5

TRUE LABELS
/K(16) * P/
CLOCK TIME = 5

R = 010001 A = 010001 C = 0100 D = 0001 F = 13 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 6

TRUE LABELS
/K(13) * P/
CLOCK TIME = 6

R = 010001 A = 010001 C = 0001 D = 0001 F = 11 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 7

TRUE LABELS
/FETCH * P/
CLOCK TIME = 7

R = 000101 A = 010001 C = 0101 D = 0002 F = 15 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

LABEL CYCLE 8

TRUE LABELS
/K(15) * P/
CLOCK TIME = 8

R = 000101 A = 010001 C = 0101 D = 0002 F = 00 G = 1 000100 = 010001
000101 = 001001 000102 = 000000 000103 = 000000 000104 = 000000

图 8.14 硬联控制计算机的部分模拟测试输出

指令程序。接着由输出例行程序指明在模拟时需把被模拟机器的哪些状态打印出来。而后经加载例行程序，把用被模拟机器指令系统写的测试程序及有关数据调进来。在模拟时，经模拟例行程序，对调进来的测试程序的每一条指令，按翻译好的 CDL 文本进行解释。并把解释结果（相当于在被模拟机器上执行其测试程序的结果）按输出例行程序的要求打印出来。

显然，被模拟机器的每条指令的执行都是在宿主机上模拟实现。例如，按图 8.6 的 CDL 文本的要求，开始应执行开关操作，但实际上开关并不存在，只不过是把宿主机中定义为 POWER 和 START 的存贮单元置一个值，代表 POWER 和 START 开关处于“ON”状态。又如，对“取指”操作，当然也不是宿主机的“取指”操作，而是由宿主机主存中定义为被模拟机器的 MEMORY 的某一存贮区内取测试程序中的一条指令送至定义为 R 寄存器的存贮单元内。

这里讲的模拟测试和一般模拟的差别还在于后者只能打印出每条指令的执行结果，而前者还能打印出被模拟机器的每拍执行结果。从而更能由打印结果判定机器的设计正确性，这是采用硬件描述语言的一个显著好处。

微程序控制计算机的模拟测试：

```

$ TRANSLATE

* MAIN
.....
.....
END
    } CDL 描述文本

$ SIMULATE
* OUTPUT LABEL(1) = POWER, START, R, F, A, C, G, D, H,
          M(101), M(102), M(103)
* SWITCH 1, POWER = ON
* SWITCH 1, START = ON
* LOAD
  CM(0-4) = 016000, 011200, 011100, 001020, 001400,
  CM(5-11) = 001040, 001010, 001004, 011202, 001001,
  M(0-4) = 100100, 010101, 040102, 070000, 030006,
  M(5-11) = 050003, 040103, 020102, 060000, 110000,
  M(100-103) = 000001, 100001, 000000, 000000
* SIM 112, 3
    
```

这里给主存 M 加载的是与表 8.6 相类似的试测程序；而给微程序控制存贮器 CM 加载的是表 8.5 的微指令。这里的输出是按标号周期进行的。图 8.15 给出它的部分模拟输出结果。

1.3 交互式计算机图形语言

很多硬件描述语言，如前述 CDL 是文本式描述语言。将自然语言陈述的设计要求，变成这种文本式设计语言的描述还是相当麻烦的，特别是当设计要求比较复杂时。为了解决这个问题，有些设计者在研究，能否按设计要求先在图形终端上用图形式语言画出寄存器配置

SWITCH INTERRUPT

POWER=ON
START=ON
POWER=ON
D=0000

START=ON
H=00 R=000000
000101=100001
F=002000
000102=000000
A=400000
000103=000000
C=0000
G=1

LABEL CYCLE 3

POWER=ON
D=0000

CLOCK TIME=1

TRUE LABELS
/F(10)*P(2)/
START=ON
H=00 R=000000
000101=100001
F=016000
000102=000000
A=400000
000103=000000
C=0000
G=1

LABEL CYCLE 4

POWER=ON
D=0000

CLOCK TIME=2

TRUE LABELS
/F(6)*P(0)/
START=ON
H=00 R=100100
000101=100001
F=016000
000102=000000
A=400000
000103=000000
C=0000
G=1

LABEL CYCLE 5

POWER=ON
D=0001

CLOCK TIME=2

TRUE LABELS
/F(7)*P(1)/
START=ON
H=10 R=100100
000101=100001
F=016000
000102=000000
A=400000
000103=000000
C=0100
G=1

LABEL CYCLE 6

POWER=ON
D=0001

CLOCK TIME=2

TRUE LABELS
/F(10)*P(2)/
START=ON
H=10 R=100100
000101=100001
F=011202
000102=000000
A=400000
000103=000000
C=0100
G=1

LABEL CYCLE 7

POWER=ON
D=0001

CLOCK TIME=3

TRUE LABELS
/F(6)*P(0)/
/F(21)*P(0)/
START=ON
H=00 R=000001
000101=100001
F=011202
000102=000000
A=000000
000103=000000
C=0100
G=1

LABEL CYCLE 8

POWER=ON
D=0001

CLOCK TIME=3

TRUE LABELS
/F(13)*P(1)/
/F(11)*P(1)/
START=ON
H=00 R=000001
000101=100001
F=011202
000102=000000
A=000001
000103=000000
C=0100
G=1

LABEL CYCLE 9

POWER=ON
D=0001

CLOCK TIME=3

TRUE LABELS
/F(11)*P(2)/
START=ON
H=00 R=000001
000101=100001
F=016000
000102=000000
A=000001
000103=000000
C=0001
G=1

图 8.15 微程序控制计算机的部分模拟测试输出

表 8.7 信息流元件及调用命令

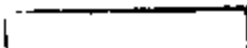
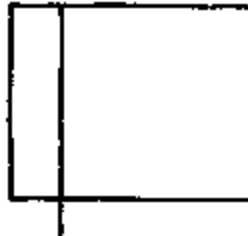

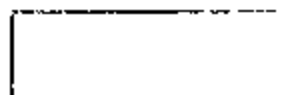
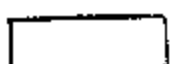

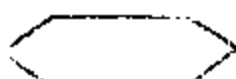
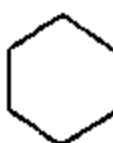
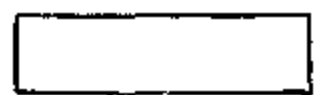
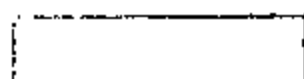







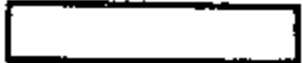

键 盘 命 令	计 算 机 回 答	含 义								
R		寄 存 器								
M		存 贮 器								
I		信息流通路								
F		功 能 元 件								
C		控 制 元 件								
L		控 制 流 线								
E		译 码 器								
B	<table border="1" data-bbox="732 1584 974 1777"><tr><td>ADD</td><td>SUB</td></tr><tr><td>MUL</td><td>DIV</td></tr><tr><td>REM</td><td>OR</td></tr><tr><td>AND</td><td>XOR</td></tr></table>	ADD	SUB	MUL	DIV	REM	OR	AND	XOR	二元运算部件
ADD	SUB									
MUL	DIV									
REM	OR									
AND	XOR									
U	<table border="1" data-bbox="743 1852 984 2059"><tr><td>NOT</td><td>COM</td></tr><tr><td>LS</td><td>RS</td></tr><tr><td>LC</td><td>RC</td></tr><tr><td>CU</td><td>CD</td></tr></table>	NOT	COM	LS	RS	LC	RC	CU	CD	一元运算部件
NOT	COM									
LS	RS									
LC	RC									
CU	CD									
K		时 钟								
S		部分寄存器								
J		组 合 逻 辑								

表 8.8 控制流元件及调用命令

键 盘 命 令	计 算 机 回 答	含 义
F		功 能 块
G		GO-TO 线
C		控 制 信 号
L		控 制 流 线
D		判 定 块
E		译 码 块
O		启 动 块
1		结 束 块
2		终 止 块

图, 控制顺序图和状态图, 然后由机器自动形成硬件描述语言文本, 并进行模拟校验, 从而使计算机辅助设计更向前进一步。美国 S. S. Ching 等人就研究出了用于此目的的交互式图形语言, 称为 FLOWWARE。实际上, 它是在普通文本式语言的翻译、模拟上之, 再加一级图形式语言的翻译; 它还能将文本式模拟结果经由图形处理器转换成图形式输出。

FLOWWARE 系统主要包括称为 IDDP (Interactive Digital Design Assistance Package) 的交互式数字辅助设计软件包, 一台能处理图形式输入的预处理器和一台能将文本式模拟结果变换成图形式输出的图形处理器。IDDP 用的就是上节讲的 CDL 语言的子集, 它包括一个交互式源语句翻译程序和一个交互模拟程序。IDDP 只能接受文本式设计描述, 所以在 FLOWWARE 系统中加了一个预处理器, 它能把从图形终端收到的图形式输入翻译成 IDDP 能接受的文本。模拟操作是由图形处理器控制, 并能将模拟结果转变成图形, 在图形终端上显示出来。实际上, 这种加予处理器和图形处理器终端的技术也可以适用于其它文本式描述语言, 使之能具有图形式语言的输入/输出。

FLOWWARE 从信息流和控制流两方面去描述数字系统。信息流包括存贮器、寄存器、部分寄存器、时钟、组合逻辑、控制, 译码器以及其它一些硬件; 在它们之间画的带箭头的连接线表示数据通路。信息流的描述基本上表达了数字系统的硬件组成结构。

控制流是用控制流程图指明该数字系统的控制算法。基本的控制机构包括功能块、判定块、译码块、启动块和终止块等。

对寄存器的操作用 IDDP 型语句写在功能块内。表 8.7 和表 8.8 分别列出信息流和控制流所用的元件符号及其调用命令。使用者可用键盘命令要计算机在图形终端的光标所示处画出需要的元件。多数元件都带有与之有关的文本式描述。这些文本式语句可以定义元件的名字和一些执行动作。另外, FLOWWARE 还提供编辑能力, 使用者能增加、修改或删除元件和相应的文本。它还有分窗口能力, 允许使用者指明一个设计需要用几个窗口 (页) 来描述。

表 8.9 运算助记符

一元运算		二元运算	
助记符	含义	助记符	含义
NOT	逻辑非	ADD	加
COM	2 的补码	SUB	减
LS	左移一位	MUL	乘
RS	右移一位	DIV	除
LC	循环左移一位	REM	余数
RC	循环右移一位	OR	或
CU	计数加 1	AND	与
CD	计数减 1	XOR	异或

下面举例说明如何应用这种人机对话式的图形语言来描述和模拟数字系统。例如, 我们要设计一个七位寄存器的串行奇偶位形成电路, 就可以在图形终端上分两步画出图 8.16 的

信息流状态图和图 8.17 的控制状态图。

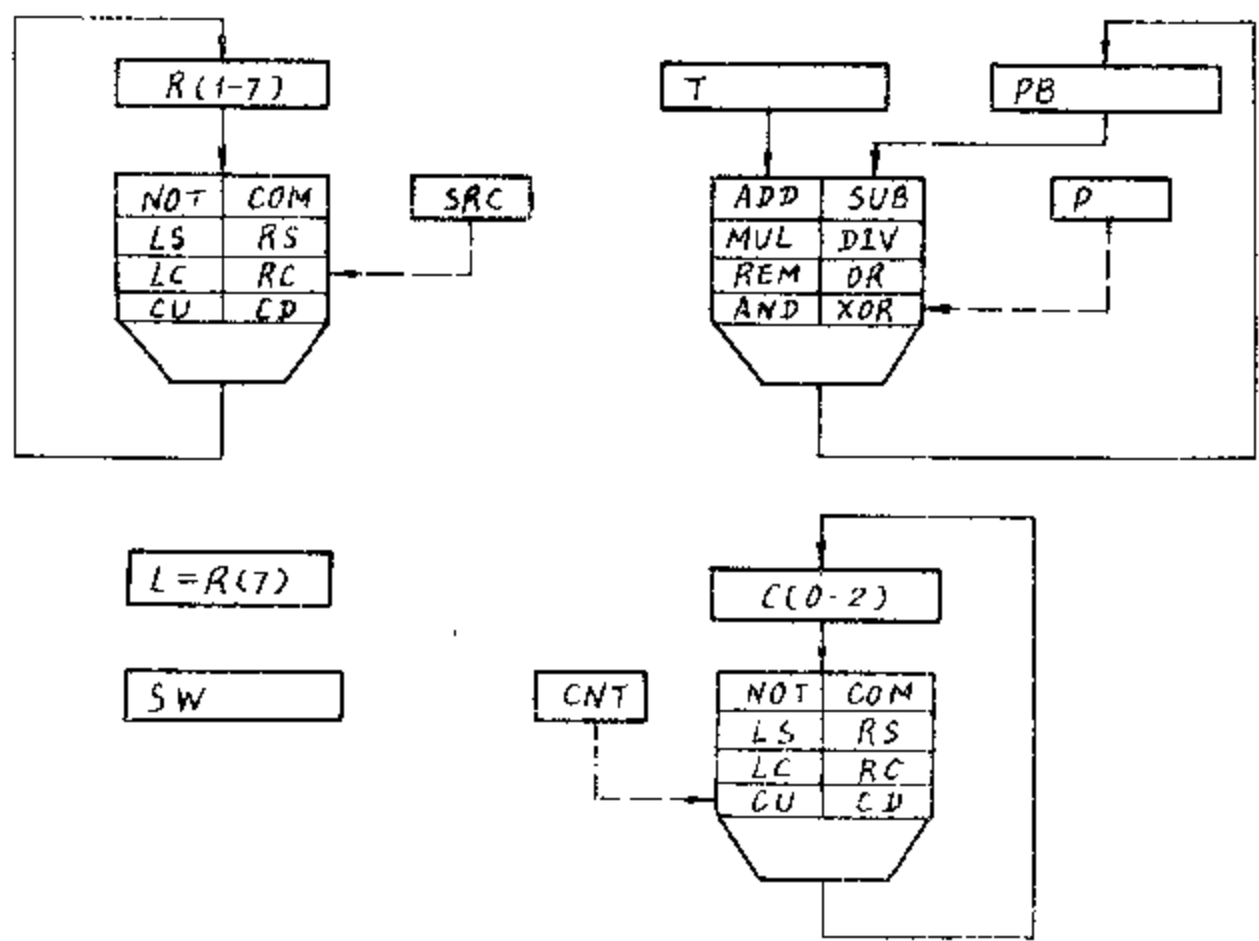


图 8.16 信息流状态图

图 8.16 是列表 8.7 列出的信息流元件指明该串行奇偶位形成电路的硬件组成。在引用表 8.9 的一元或二元运算时，使用者必须用控制流线从一控制元件指向所选用的运算。这样，当控制元件动作时，就对送入的操作数进行这种选定的运算。图 8.17 是用控制流元件以流程图的形式，描述了该奇偶位形成电路的动作序列。动作沿着 GO-TO 线指出的方向顺序进行。其中控制信号 SW 的作用好像一个开关，在模拟时它可打开或阻塞一个控制流路径。如果在模拟时 SW 值为“1”，图 8.17 的控制流程可以被执行，否则被封锁。在功能块里可以写上一个或几个 IDAP 语句，当模拟进行到某一功能块时，块里所有 IDAP 语句都同时被执行。在执行功能块中的“DO name”语句时，就引起信息流状态图中“name”控制元件动作。例如，执行“DO CNT”语句时，就引起信息流状态图中的 CNT 控制元件动作，从而实现计数器 C 的内容加 1 操作(CU)。

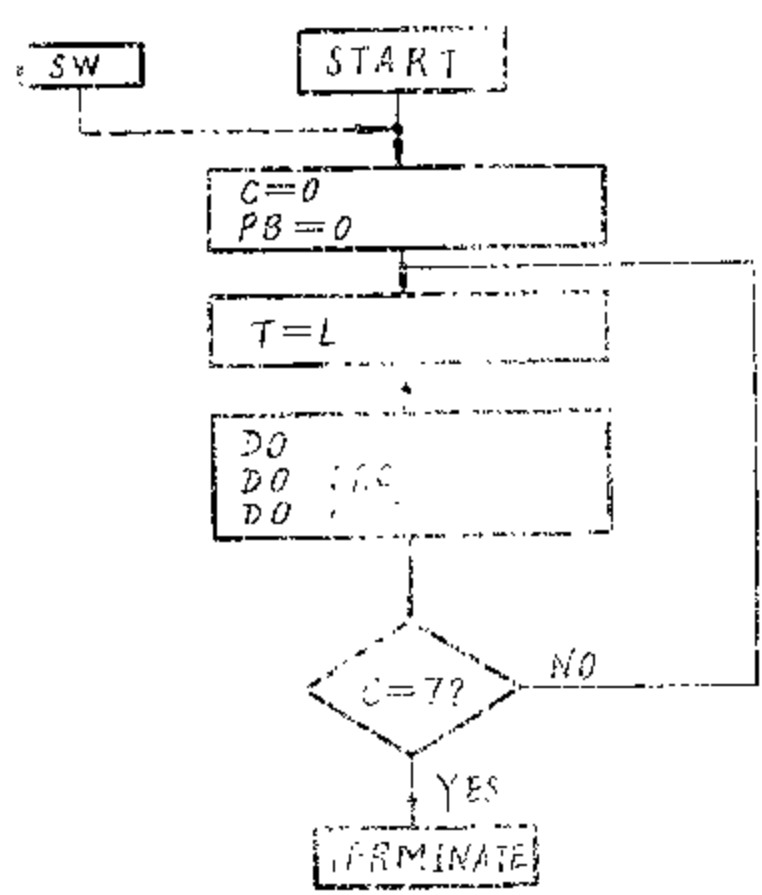


图 8.17 控制流状态图

结合图 8.16 的信息流状态图和图 8.17 的控制流状态图，就可以看清该 7 位串行奇偶校验位形成电路是如何工作的。图 8.17 控制流状态图指明模拟从控制信号 SW=1 时开始。在控制流路径上的第一个功能块，将计数器 C 和奇偶校验位寄存器 PB 置为“0”。其余控制流程构成循环。在循环的第一个功能块里，将寄存器 R 的最右一位逻辑值经 L 寄存器送至

寄存器 T。下一功能块使控制信号 P, SRC 和 CNT 同时工作,引起信息流状态图中三个运算操作:完成求 T 和 PB 的异或,其结果放入 PB 中;使 R 寄存器内容右移一位;给计数器 C 加“1”。在循环结束的判定块中,根据 C 的计数值是否等于 7 决定控制流是走出循环还是返回循环。这样循环执行 7 次后,在 PB 中就得到 R 原存内容的奇偶校验值(例如,若 R 原为 1011010,执行结果 PB=0)。走出循环控制流遇到终止块,它指明停止模拟。如果遇到的是结束块,只表明该分支路径结束,模拟还可以从其它分支路径继续进行。

FLOWWARE 系统的模拟程序可执行三种类型的模拟:常规模拟、分段模拟和步进模拟。

常规模拟:模拟从启动块开始,遇到终止块停止。模拟过程一次执行完。

分段模拟:使用者通过在控制流路径上设置一些断点,将模拟过程分几段执行。FLOWWARE 只允许在控制流状态图中的功能块处设立断点。模拟可以从启动块或最近一个断点处开始,到下一个断点或遇到一个终止块为止。

步进模拟:每次启动只能模拟一个功能块,这时使用者可观察到中间结果,并接着可让模拟继续进行。

后面这两种模拟方式允许使用者仔细观察和研究模拟过程。另外,系统终端还有对控制流路径上正在工作的块加强显示亮度的功能,用于示踪模拟的进程,并可在信息流状态图中,在即将动作的元件上标以星号以便于观察。

在模拟时,使用者可用两类键盘命令直接控制图形处理器的操作。表 8.10 的第一类键盘命令是直接作用于信息流元件。用户在使用这类命令之前应先将光标移入指定的元件内。有些命令可能还需要在光标处打入文本式字符或数字。例如,在本例中,R 寄存器的初值就需用这种命令置入。表 8.11 的第二类键盘命令是控制模拟方式与进程的一些子命令,它们的名单显示在图形终端屏幕的底部,可供选择调用,不必使用者记忆它们。

表 8.10 第一类键命令

键 命 令 符	
S	置元件值
D	显示元件值
C	建立主时钟
E	使能波形收集。即收集指定元件在模拟时的数值变化,以时间波形图显示
R	停止指定元件的波形收集
B	在指定的功能块之前设置一个断点
K	从断点表中去掉指定的功能块
W	重画当前显示的元件,但不列出元件中的文本内容
M	显示并改变存贮单元的内容

综上所述,硬件描述语言确实是很有用的辅助设计计算机(数字系统)工具。它不只是研究、制造、使用、维护人员之间方便交流的工具,同时也适合作为教学的工具。描述语言

表 8.11 第二类键盘命令

键 盘 命 令	功 能
(1) SIMULATE	调用模拟程序。如不加任何说明就执行常规模拟。 如果打入B命令符则按分段方式模拟。
(2) GO BACK	使用户能返回到之前在模拟过程中曾遇到过的一些断点， 模拟环境也恢复到当时的相应情况
(3) STEP	选用步进方式模拟，只模拟一个功能块
(4) SIZE	可以改变画面的大小，可选全屏幕显示，多窗口显示，或将屏幕分为两 半，一半描述信息流，一半描述控制流
(5) DISPLAY	显示当前在终端上所有元件的逻辑值
(6) WAVE	在终端上画出模拟过程中的时间波形图
(7) CLOCK	使系统的动作是定时的或非定时的
(8) PHASE	选择被描述的状态是信息流或是控制流
(9) MOVE	使能选择新的显示窗口
(10) REDRAW	重画。为获得描述的新拷贝
(11) CLEAR	置 FLOWWARE 于起始状态
(12) EXIT	从图形控制处理器退出。 从此，用户可以开始新的描述或终止设计

是模拟技术的重要工具。而模拟技术又是下节要讲到的性能评价研究中的重要手段。总之，硬件描述语言是一个很值得研究的课题，在各国都作了很多研究工作，已研究出种类繁多的描述语言。新的硬件描述语言还在不断地增加，但真正应用到实际辅助设计中的并不多。目前的计算机设计，大多数还是画逻辑图的传统老方法。处于这种状况的原因是：

(1) 还没有哪一种硬件描述语言能单独地描述系统各级和各个方面的特性，又能概括整个设计过程的各个阶段；

(2) 各级各阶段所用的硬件描述语言在语法和语义上差异很大，甚至毫不相关；

(3) 只有极少数硬件描述语言是已形式化定义的；

(4) 已配有翻译程序和模拟程序的只有极少数几种；

(5) 几乎都是用字符串而不是用图形来描述；

(6) 还没有能有效地利用硬件描述语言而又易于理解的硬件和固件设计方法。

看来，需要研究这样一种形式化硬件描述语言，它既具有能适用于描述从系统构成、特性到各级、各个方面的公共核心语法和语义；又能根据特定的设计任务，从这个核心语法和语义推导出适用于这个任务某个级的子语言。这可能是促进硬件描述语言能更快普遍地应用于工程实践的正确途径。

§2 性能评价

下面谈谈计算机系统的性能评价。

2.1 前言

前面我们是从自动设计的要求引出研究性能评价的必要性，但同硬件描述语言一样，自有计算机以来，一直存在如何对其性能进行评价的问题。最初，计算机的设计目标是在机器能正常运行的前提下使速度尽可能快。这个前提可以用可用性或可用率来表征。前已讲过，可用率指的是：

$$\frac{\text{可用时间}}{\text{可用时间} + \text{故障时间} + \text{维护时间}}$$

可用率是衡量机器好、坏的重要标志。可用率过低的机器是没有使用价值的。提高可用率的方法是与提高机器的可靠性的方法紧密相关的。然而，可用率不属于这里讲的机器性能，这里讲的性能指的是机器在可用状态，即正确运行状态下的性能。长期以来，衡量机器性能的主要标志是速度的高低。问题是速度本身用什么来衡量。

最初的计算机不只是它们的结构相同（如图 1.14 所示），而且它们的组成也几乎相同，因此，机器的主频能够反映机器的速度，机器的主频越高，其性能越好。但是，当相同结构的机器采用不同的组成时，机器主频就难以确切反映机器的运算速度了；例如，具有相同主频的二台机器，一台设计成完成一条加法指令需要 10 拍，另一台则改进为仅需 8 拍，显然，在相同主频下后一台机器的性能就好些；又如，具有相同主频的二台机器，其主存周期可能不同，自然是主存周期短的，其运算速度快。因此，倒不如用指令的运算速度来表征为好。

可是，即使是早期的机器，指令系统也有几十条，各条指令的运算速度又不相同，那怎么来衡量机器的指令运算速度呢？那时的机器，因为运算方法基本相同，加法指令的运算速度可以反映出乘、除法等其它算术运算指令的速度。因此，最初就是用加法指令的运算速度来表征机器的速度，其计量单位由 KIPS（每秒运算千次）发展到 MIPS（每秒运算百万次）。这个指标至今还在使用，不过后来不一定指的是加法指令而是指令系统中执行最快的那条指令的 MIPS。用单种指令的 MIPS 表征机器的运算速度有严重缺陷，很快就有了各种修正。五十年代我国采用的下述等效指令速度法就是一例。它认为不应只考虑加法指令的速度，还应考虑其它算术运算指令的速度，而且，各类指令占整个程序执行时指令的比例分别取为：

加（减）法指令	25%
乘法指令	15%
除法指令	5%
其它指令	55%

若这些指令的执行时间（若有浮点指令时则取浮点运算指令的）分别为 $t_{\text{加}}$ 、 $t_{\text{乘}}$ 、 $t_{\text{除}}$ 、 $t_{\text{其它}}$ ，则等效指令执行时间为：

$$T = 0.25 \times t_{\text{加}} + 0.15 \times t_{\text{乘}} + 0.05 \times t_{\text{除}} + 0.55 \times t_{\text{其它}}$$

机器的运算速度即为 $1/T$ 。

从表面看，用等效指令速度表征机器运算速度要比只用加法指令速度表征要全面准确些，然而，上述比例算出来的机器速度往往偏离实际。更由于在很长一段时期内领导思想和

政策上的片面追求运算速度，这种速度算法对我国计算机的评价和设计造成了严重恶果。在实际程序中，各类指令根本不是上述那种比例。首先是算术运算指令的比例远比上述 45% 要低，一般只占 20% 左右；其次，就算术运算来讲，乘、除法指令实际所占比例也远比上述低，尤其是浮点乘、除法的比例更低，分别仅为 3~4% 及 1~2%，比上述的 15% 与 5% 低。这样，如果按上述不正确的比例要求来设计机器的话，势必片面地去追求提高算术运算的速度，尤其是片面地去追求提高浮点运算速度和乘、除法速度。这往往造成机器的畸型发展，使得 CPU 中为提高浮点速度和乘、除法速度所设置的那部分（往往是庞大的）设备时间利用率极低。可惜的是五十年代提出的这种错误的比例，竟长期（直至七十年代）为我国计算机设计和使用部门所采用。由此也可看出研究计算机性能评价的重要性。

对于等效指令速度法，如何选取符合实际情况的比例是很重要的。六十年代，有不少人致力于根据实际机器的运行统计来确定各种指令的比例，其中 Gibson 对在 IBM-7090 机器上运行的程序进行过统计和分析，在六十年代末提出了一个比较符合实际的比例，得到比较广泛的使用。此外，Flynn 对在 IBM360 机器上运行的科学计算和工程计算程序进行统计、分析，在 1974 年也提出了一个比例。这两个比例列于下表：

指 令 类 型	Gibson %	Flynn %
取/存	31.2	} 45.1
变址	18	
转移	16.6	27.5
比较	3.8	10.8
定点		7.6
加/减	6.1	
乘	0.6	
除	0.2	
浮点		3.2
加/减	6.9	
乘	3.8	
除	1.5	
位移/逻辑运算	6.0	4.5
其它	5.3	1.3
	100.0	100.0

可见，取/存、变址、转移、比较、其它等非算术运算指令数所占的比例达 74.9% (Gibson) 或 84.7% (Flynn)；而浮点运算（包括加、减、乘、除）却只占 12.2% (Gibson) 或 3.2% (Flynn)。

随着计算机结构的发展与复杂化，不论采用什么样的比例，等效指令速度终究反映不了从结构上提高机器性能的某些重要改进，例如，I/O 处理机的采用，使 I/O 操作与 CPU 运算可重迭运行，大大提高了机器的实际速度，可是结构上的这种改进从等效指令速度却反映不出来。而且，等效指令速度也反映不出字长不同的影响，不同字长的机器其实际解题速度往往会差别很大。例如，二台具有相同等效指令速度的机器，其中一台的字长为 32 位，另一台的为 16 位，当要求 32 位精度的运算时，字长只有 16 位的机器，就得采用双倍长运算（用

子程序方法)，其实际解题速度当然要比字长为 32 位的差得多。举这两个例子是想附带说明为什么不能用等效指令速度来对结构不同的我国 DJS-100（字长为 16 位，无 I/O 处理机）与 DJS-200（字长为 32 位或 64 位，240、260 有独立的 I/O 处理机）系列机进行比较。除了字长不同以及是否有 I/O 处理机外，这二个系列机还有其它结构上的差别，例如数据类型、中断机构等等的差别也是等效指令速度所反映不了的。再有，对采用流水结构的机器，在第三章讲过，实际运算速度往往比计算出的等效运算速度要低得多（会低 2~3 倍），这是因为等效运算速度是按正常顺序流水时，各类指令的执行时间算出来的。而实际运行时，碰到各种相关时，机器速度就会显著下降。

如果再联系到系统软件，等效指令速度法就更显出其不适用性。例如，同样一条高级语言语句或同一段高级语言程序，由于机器指令系统的不同，就可能对应不同长度的机器语言目的程序。相同等效指令速度的不同机器，由于机器指令系统设计的好坏，使高级语言程序的编译时间和所编译出来的目的程序的执行时间可能不同，甚至差别很大。又例如，多道程序运行的计算机系统，由操作系统把一道程序切换到另一道程序所需的辅助操作量以及所化的时间直接影响到机器的处理能力，在第四章讲过，这和系统结构的设计好坏很有关系，但这也是等效指令速度法难以反映出来的。还有，编译程序与操作系统本身也是由机器指令构成，不仅是它们对应的机器指令执行比例与应用程序的不同，就是编译程序与操作系统这两者所对应的比例也不同。

因此，六十年代末，七十年代初开始，有不少人致力于研究应怎样对包括软、硬件的计算机系统，而不仅是计算机裸机本身，进行更为准确的性能评价。

“核心 (Kernel) 程序”法是研究得较多，也是用的较广的一种。它的出发点是把应用程序（有的也包括系统软件）中用得最频繁的那部分核心程序作为评价计算机系统性能的标准程序。直接在不同机器（或是在其模拟器）上运行，测定其执行时间，以此作为性能评价的依据。这种方法当然要比等效指令速度法更接近于实际的运行状况，更能反映机器真正的速度性能。这种程序一般用汇编语言编制。当然核心程序不能只是一个，而需多个，并且要能反映软件和硬件的主要功能。

“典型程序 (Benchmark)”法则是核心程序法的发展。它是用包括有输入/输出操作的典型程序作为性能评价的标准程序。它们一般用高级语言编写，使之还能反映编译程序的运行状况和编译的效能。另外，宜于采用多个典型程序混合运行，这样就还能反映操作系统的多道程序管理以及存贮层次的设计好坏。在实际使用中核心程序与典型程序有时并无严格区别，这两个名词往往可以混用。

这两种方法可以统称为标准程序法。其缺点是难以选择真正有代表性的核心程序与典型程序，因为往往随不同的应用范围而不同。此外，若是用汇编语言编制标准测试程序，对不同机器就需为之分别编制，比较麻烦费时。就是用高级语言编制也不能不加修改地使用于各种机器。

标准程序法是要联系机器的实现（组成）由实测时间来评价，这不利于用来评价各种系统结构（如各种不同的系列机的结构）的性能，因为对同一种结构（一般，一个系列机采用一种结构）采用不同的实现方法（如系列机内各档），本是可以具有不同的速度。如果只从实测时间，如何判定出各种系统结构的设计好坏呢？我们在 § 2.6 介绍系统结构的一种评价方法

时就采用了不联系实现的评价法。这种评价系统结构的方法正是反映了研究系统结构的特点。

性能评价是研究计算机系统的一个重要方面。要想得到满意的系统，设计和评价都是不可缺少的环节。但是，过去我们对性能评价研究的较少，在实际工作中吃了很多亏。一提到评价，可能被认为是等一个系统设计好后才来评价它的性能好坏。这种认识是片面的。实际上从系统设计开始，甚至在设计开始前就应进行性能评价工作，否则设计会是盲目的。当然，对已存在的计算机系统自始至终仍然有性能评价的问题。例如，计算机系统的用户在购置、租用、更换、改进系统时，都必须通过一系列评价过程才能正确进行。如果作决定时没有评价技术作依据，将会造成不应有的浪费。

计算机系统的性能就是计算机系统的使用价值，价值总是应与价格相联系，所以，对计算机系统性能的分析总要联系到计算机系统的价格，这就是性能/价格比的概念，显然，这个比值应当越大越好。

性能既是价值，它的评价就具有某种主观性。也就是说，不同的人，从不同的角度会有不同的评价标准；但是，这些标准或指标终究要受客观的检验，当然也会在使用中逐步取得一致，并尽可能定量化。当然，随着计算机系统的不断发展，它的评价标准或性能指标也在发展变化。

我们所说的性能评价不是等获得（设计的或购买的）一个系统之后，鉴定一下它有无应该具有的功能。这种功能的有无只是性能评价的先决条件，也是最容易判定的（譬如，一个放大器它首先应该能放大有用信号，否则还有什么性能可评价）。这不是性能评价所研究的目标。性能评价指的是在一个系统满足规定的功能前提下，分析估评该系统工作的好坏。对一个给定的系统作出恰当的评价并不是一件容易的事。譬如对小汽车，一辆速度快、便宜、易维修，但费油、寿命短；而另一辆速度较慢、省油、寿命长、舒适，但价格较贵，难维修。你说哪一辆好？这就要看评价者的观点。你若需要赛车，速度快则是第一重要的指标；如果作为日常交通工具，省油、耐用、舒适相对于速度可能更重要。事实上不同的评价者对性能指标都加了不同的权值。一个系统对人家好，对你却不一定好。对计算机系统更是如此。

对计算机的研究包括很多方面。例如，计算机网、计算机系统、系统结构、计算机设备、程序设计、语言和语言翻译等等，它们有着各种不同的功能与特点，也有着不同的评价方法和要求。提出评价要求的人员也是来自各个方面。例如，有生产厂家、管理人员、维护人员和用户等，他们都有着各种不同的主观要求。目前所说的性能评价只讨论计算机系统本身（主要包括系统结构、计算机设备和操作系统）的评价。对于复杂的计算机系统进行评价是有很多困难的。首先，有关计算机系统性能的描述有很多，哪些是应该和必须评价的重要指标，还没有统一的标准。而且，有些性能是无法用数学方法定量的描述，因此很难比较（例如便于使用、易维修等）。还有些指标虽然可定量描述，但也难于判定（例如寿命等）。当然，评价工作的困难还在于要对一个系统作出正确评价，必须要求评价者对系统了解的很清楚，涉及到的知识面很宽，包括硬件和软件方面的。

性能评价技术已发展成为一个独立的学科，详细讨论评价技术需要另开一门“计算机系统性能评价”课。我们这里简单地介绍评价计算机系统的一般观点、步骤和方法。

2.2 评价计算机系统的步骤

性能评价的研究不同于连续性的系统监督工作。评价研究一般都有时间限定，需要在较

短的时间内完成，而且所要评价的系统不一定是已有的。一般的评价研究工作可能包括以下五个步骤：

- (1) 分析是否有评价的需要；
- (2) 明确评价的目的；
- (3) 制定评价研究的计划；
- (4) 实现计划；
- (5) 判定评价结果。

它们在一个完整的性能评价研究工作中是不可少的，但它们不是截然分开，这五个步骤是一个反反复复的过程。为了将它们之间的相互关系说得更清楚，可用图 8.18 中的关系图来表示。

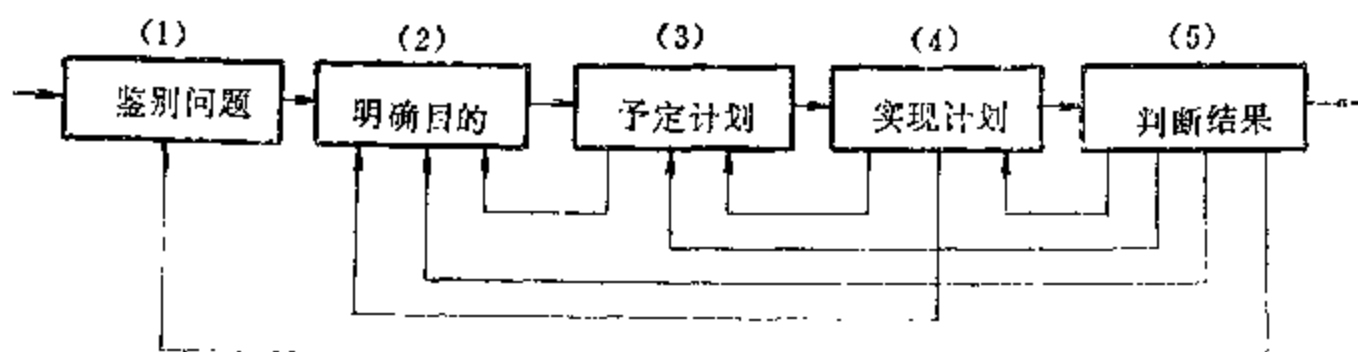


图 8.18 性能评价研究的步骤

第 (1) 步对提出的评价要求和问题进行鉴别，分析是什么类型的问题。有的计算机系统工作效率不高不一定是计算机系统本身有什么大毛病。有时只是时间片选得不合适，有时只是磁盘文件存贮管理的不合理，使得一个磁盘经常有排队，而另一个却总闭着。解决这类问题并不需要增加、改换任何设备，只需稍稍调整一下就可解决。如果不经仔细分析就乱加改动，只会白浪费时间和金钱。第 (2) 步是很重要的，目的不明确，其它工作就无法进行。另外，在制定计划之前要仔细估算所需费用及评价结果的利益。有的评价研究在经济上是不合适的。譬如，对单机生产的一般机器就没有必要化费很多的经费和时间去进行详细的性能评价。第 (3)、(4)、(5) 步一般是需要经过多次反复的。对预定的计划进行试验，结果不能满意就要重新制定计划再试，直至得到满意的结果为止。当然，如果评价者对系统内部结构和特性了解得很清楚，就可能减少这种反复次数。在评价过程的每一步都要仔细阅读有关手册，查阅逻辑图、流程图和程序表；询问操作员和其它有关人员，访问用户，测量负荷参数，收集有关系统使用数据等等。另外，还应参考已有的评价报告，学习别人的经验。

2.3 计算机系统的性能指标

性能指标是对系统性能和特征的描述。我们前面已经讲过，系统性能有些是不能定量描述的。我们这里只讨论能比较客观地、定量地测定的一些性能指标。

计算机系统中最常用的，可定量描述的性能指标列于表 8.12。对信息量的表征可能是五花八门的（例如，有作业、程序、过程、作业步、任务、指令等等），目前对系统处理信息量的量度还没有标准化。另外要特别注意，这里的所有性能指标没有哪一项不与工作负荷以及系统特性（例如，结构组成、机器语言、程序语言等）有关。评价者必须清楚是在什么

表 8.12 可量度的性能指标

指标类别	指 标	一 般 定 义
工 作 量	吞吐率 工作速率 工作能力 (最大吞吐率) 指令执行速率 数据处理速率	吞吐率指的是系统在单位时间内能处理的信息量
响 应 性	响应时间 周转时间 反应时间	响应时间指的是从给定系统输入到出现对应的输出之间的时间间隔
利 用 率	硬件 (CPU、主存、I/O 通道、I/O 设备) 的利用率 操作系统的利用率 公用软件 (如编译程序) 的利用率 数据库的利用率	利用率指的是在给定的时间内, 系统某一部分的实际使用时间所占比例

样的系统和什么样的负荷情况下测得的性能指标, 否则是没有意义的。

下面, 我们结合交互式系统来阐述性能指标。对这种系统, 用户最关心的性能指标是响应时间。交互过程的简化时间关系表示于图 8.19。一般来说, 响应时间取决于用户输入的信息, 取决于系统特性以及在用户输入信息的同时系统正在处理的其它负荷。虽然响应时间是一个随机变量, 但在相当长的时间周期内我们可以用统计规律描述它。在这里顺便讲讲用

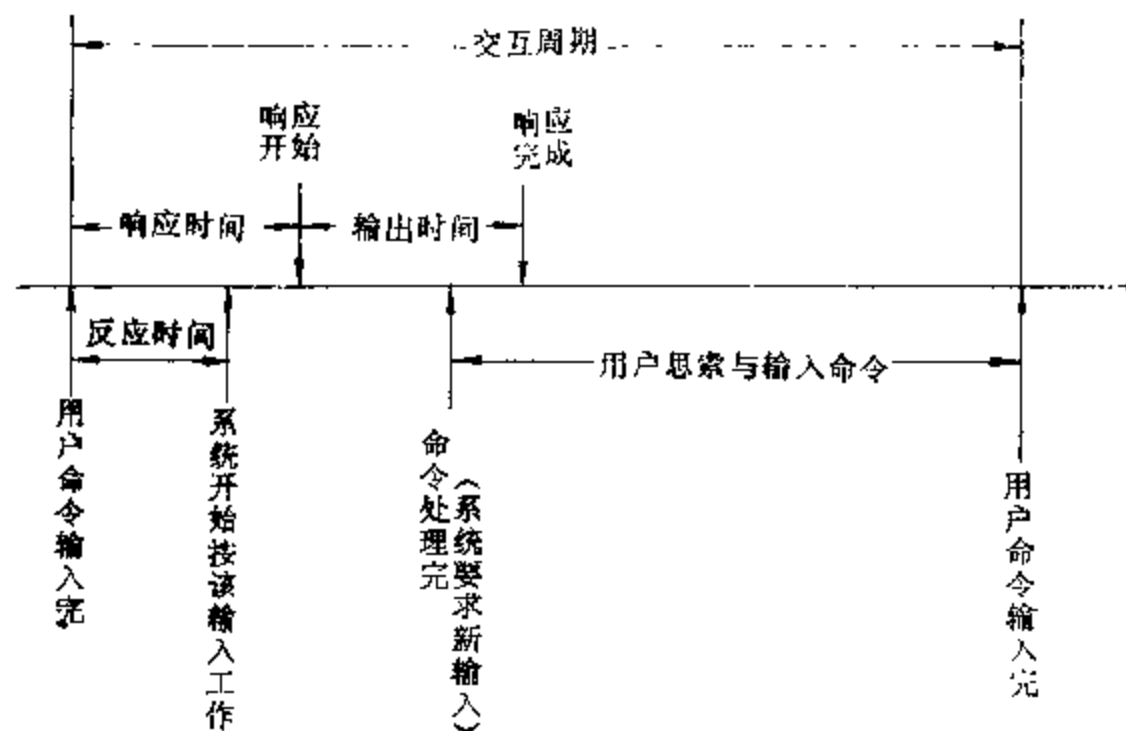


图 8.19 交互过程时间关系

用户对交互式系统的响应时间的要求。从图 8.20 可见, 响应时间在 1 秒钟以内用户最满意; 响应时间从 2 秒至 4 秒用户感觉差不多; 而响应时间超过 5 秒钟, 用户就很不满意了。

下面我们结合交互式系统的具体例子来进行一些定量的性能评价。

假设有一个交互式多道程序分时系统, 它没有虚拟存贮器。假定要执行的程序都已加载

至内存，没有申请磁盘或磁鼓的命令。用户发出的六个命令是几乎同时到达，但在处理它们的时间内再没有其它命令进入。系统是采用先来先服务、循环排队的规则。这些命令到达的次序是：一个编辑命令，跟着两个编译命令，后面又是三个编辑命令。若一条编辑命令需要1个单位的CPU时间，一条编译命令需4个单位的CPU时间；从处理一个命令到处理另一条命令的切换需占用0.2个单位CPU时间（辅助操作时间）。

若时间片取为2个单位的CPU时间，则系统处理这六个命令的时间关系图示于图8.21(a)。在时间轴上打上圆圈的数值是表示某一条命令处理完的时间。从图8.21的时间关系图可以计算出平均响应时间：

$$\begin{aligned}\text{平均响应时间} &= (1.0 + 6.6 + 7.8 + 9.0 + 11.2 + 13.4) / 6 \\ &= 8.16 \text{ 单位时间}\end{aligned}$$

$$\begin{aligned}\text{编辑命令的平均响应时间} &= (1.0 + 6.6 + 7.8 + 9.0) / 4 \\ &= 6.1 \text{ 单位时间}\end{aligned}$$

$$\begin{aligned}\text{编译命令的平均响应时间} &= (11.2 + 13.4) / 2 \\ &= 12.3 \text{ 单位时间}\end{aligned}$$

$$\begin{aligned}\text{平均吞吐率} &= 6 \text{ 命令} / 13.4 \text{ 单位时间} \\ &= 0.447 \text{ 命令/单位时间}\end{aligned}$$

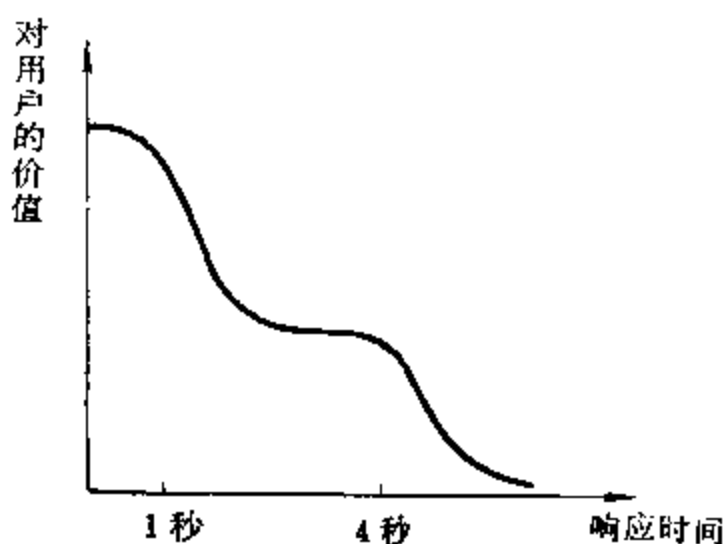


图 8.20 用户对响应时间的反应

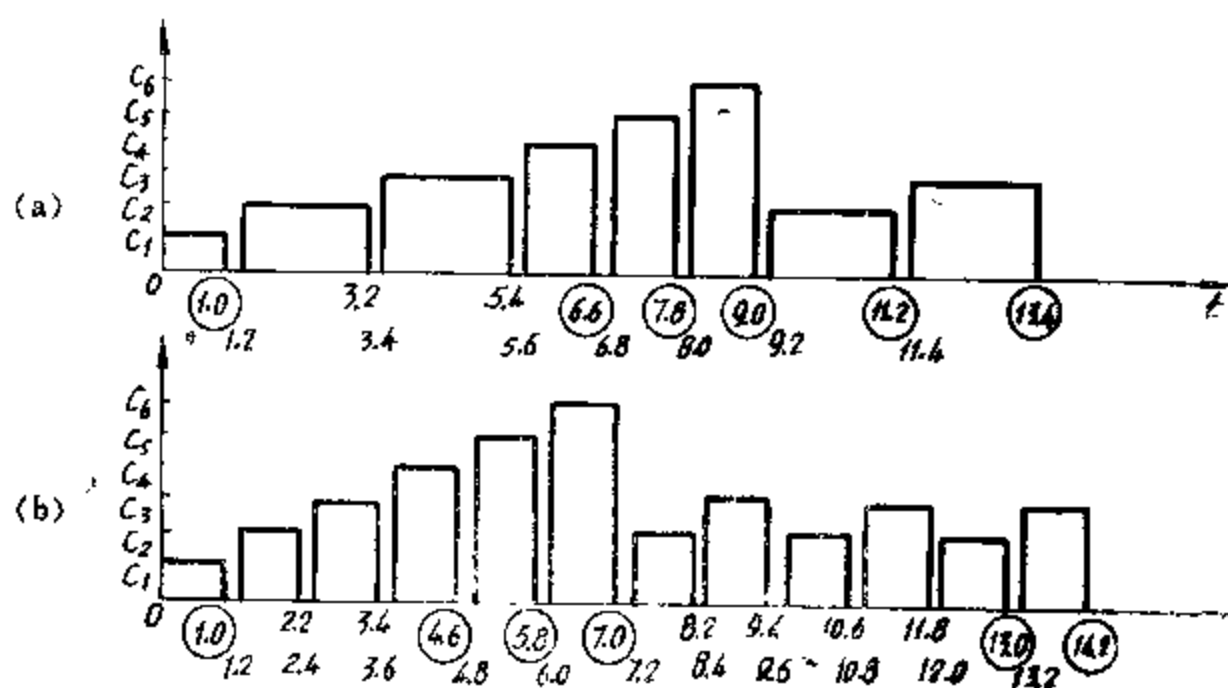


图 8.21 分时系统的命令执行时间图

如果我们将时间片改为一个时间单位，定时关系就如图8.21(b)所示。这时我们可计算出

$$\begin{aligned}\text{平均响应时间} &= (1.0 + 4.6 + 5.8 + 7.0 + 13.0 + 14.2) / 6 \\ &= 7.6 \text{ 单位时间}\end{aligned}$$

而编辑命令的平均响应时间 = $(1.0 + 4.6 + 5.8 + 7.0) / 4$

$$= 4.6 \text{ 单位时间}$$

$$\begin{aligned}\text{编译命令的平均响应时间} &= (13.0 + 14.2)/2 \\ &= 13.6 \text{单位时间}\end{aligned}$$

$$\text{平均吞吐率} = 6 \text{命令} / 14.2 \text{单位时间} = 0.422 \text{命令/单位时间}。$$

从上述计算可以看出，减少时间片对改善平均响应时间有好处（从 8.16 单位时间减至 7.6 单位时间），特别是对所需处理时间短的小作业好处更多（如编辑指令的平均响应时间从 6.1 单位时间减至 4.6 单位时间），这当然使用户感到满意。但是，另一方面却使系统的吞吐率从 0.447 下降为 0.422，这却是管理人员所不希望的。可见，一个系统的各种性能指标之间常常是密切相关的，改善其中一种指标很可能会影响到其它指标。另外还可以看出，如果输入命令到达的数量和次序有所改变，或者系统服务规则不是先来先服务的方式，那么平均响应时间和平均吞吐率都要改变。所以说，性能指标的估值强烈地依赖于系统特性及其负荷特征。

在性能评价研究中，除了评价者真正感兴趣的主要性能指标外（如上述的响应时间），还要考虑第二位的一些指标（如上述例子中的吞吐率），它们可能是评价过程中附带提出的要求。但这些第二位指标的变坏却可能是系统的某些方面性能不佳的征兆，也可能由此发现系统瓶颈的所在。

2.4 评价研究的分类

根据性能评价的目的，我们可以把评价研究分为以下三类：选择；改进；设计。

选择评价多数是由用户提出。例如，某一用户需购置一台计算机。他首先提出一系列功能要求和性能指标作为选择标准。可能同时有几家公司根据该用户要求推荐几种不同型号供他选购。这时，用户就要组织进行选择评价研究工作，通过一系列评价步骤，对提供的各种型号机器进行比较，最后选定最为满意的一种。

对一个已有的计算机系统在工作了一段时间后，若感到系统性能或效率不满意，或是拟增加一些投资以扩大系统的功能，那就可能提出**改进评价**的研究要求。通过一系列评价过程来研究已有系统存在的问题，特别是要找到系统工作的瓶颈所在。例如，一般来说，增加主存容量是会改善系统的性能。但是，如果你的系统在当前的工作负荷下其瓶颈并不在主存，而是在 CPU，那么，增加主存容量虽然费了不少钱但并不能改善系统的性能。还要注意，并不是非得给系统增加或替换一些硬件设备才能改进系统的性能；有时，不用增添任何设备，只需调整一下系统参数（例如，前述的改变操作系统所用时间片的大小，改变磁盘文件的存贮位置等），就能改进系统的性能。

设计评价研究通常是计算机系统生产厂家提出的。组成系统的部件可能是已有的，也可能还没有。如果不是已有的，就还要根据系统设计所要求的技术指标对这些部件进行评价研究。不过，设计时提出的性能指标可能没有选择研究时提的那样具体、详细，而只给出比较粗略的范围。设计评价研究很重要，它决定了所生产的计算机系统是否会有市场和具有生命力，它应该和系统的设计并行进行。

当然，性能评价的研究难于这样严格划分。例如，确定一个系统的主存容量，你可以把这看成是一个选择问题，因为主存容量一般是按模块扩展的；你也可以把它看成是改进问题，如果主存容量正是影响该系统性能的主要因素；你也可以把这看成是设计问题，如果这个主

存容量是操作系统设计者要求的主存最小容量。因此，在实际工作中这三种类型的评价研究界限有时不是很清晰的。

2.5 评价技术简述

为了进行评价研究，必须收集有关系统性能的数据，获得这些数据的方法就叫评价技术。已有的评价技术大致可分为以下两类：

- (1) 测量技术：数据是直接由系统测量得到的；
- (2) 模型技术：数据是按系统模型求得的。

当被研究的系统还不存在时，就必须采用模型技术。它又分为模拟模型和数学分析模型。

为了评价研究的方便，在研究分析系统某些方面的特性时，应将真实系统简化。尽量保留系统中与所研究问题有关的结构，去掉对研究问题影响不大的因素，建立抽象的模型，称之为概念模型，它是评价研究的重要工具。显然，同一个系统在研究其不同方面的问题时，可能抽象为不同的模型。

为表征系统的工作特性，一种方法是采用状态和状态转移的概念。系统在时刻 t 的状态可用该时刻有关的一组系统参数确定，这些参数值的变化就表征着系统从一种状态变为另一种状态。模拟模型技术采用了这种概念，它能在时间域内复制出被研究系统的状态转移序列，并且按着某种约定建立模型状态和真实系统状态之间的对应关系。这样，我们就可以用对模型特性的研究以求得性能指标来代替对真实系统的研究。虽然从原理上讲，这样得到的数据应该与在真实系统上测量的相同；但是，任何模型都带有某种假设条件，从模型上确定的数据的可信程度如何，终归还要在真实系统上去验证。所以，在某种意义上讲，测量技术仍是评价研究的根本手段。当然，对于已验证过的公认方法，仍可大胆应用，不必再去证实。

如果研究的系统特性能用数学模型表示则更为精练。数学模型只应包括一组表示系统状态转移的逻辑条件方程式，并能用数学方法求解这些方程式。这种方法叫做数学分析方法。例如，在前节例子中计算过的系统吞吐率可以表示为

$$\left\{ \begin{array}{l} \bar{T} = \frac{n}{\sum_{i=1}^n t_{CPU_i} + n_{sw} t_{sw}} \\ n_{sw} = \sum_{i=1}^n \left[\frac{t_{CPU_i}}{q} \right] - 1 \end{array} \right.$$

其中 \bar{T} 为平均吞吐率；

n 为被处理的命令个数；

t_{CPU_i} 为处理第 i 个命令需要的 CPU 时间；

t_{sw} 为从处理一个命令转到另一个命令所需的 CPU 切换时间；

n_{sw} 为 CPU 的切换次数；

q 为时间片大小。

如将前述例子中的数据代入

$$n = 6, t_{s\omega} = 0.2, q = 2,$$

$$t_{CPU_i} = \begin{cases} 1 & (\text{当 } i = 1, 4, 5, 6) \\ 4 & (\text{当 } i = 2, 3) \end{cases}$$

$$\begin{aligned} \text{则 } \bar{T} &= \frac{6}{(1 \times 4 + 4 \times 2) + \left(4 \times \left\lceil \frac{1}{2} \right\rceil + 2 \times \left\lceil \frac{4}{2} \right\rceil - 1\right) \times 0.2} \\ &= \frac{6}{12 + 1.4} \\ &= 0.447 \end{aligned}$$

与前节求得平均吞吐率完全一样。但比前述模拟分析方法简练得多。

显然，用任何间接测量方法获得的数据来进行评价，其结论应经过验证确认，这往往是困难的。为此，可以采用不同的两种方法，并对所求得的结果进行比较，以提高可信度。

上面初步介绍如何对计算机系统进行性能评价，下面则结合具体例子讲述如何对系统结构进行评价。

2.6 对系统结构的评价

下面结合美国的一个系列机系统结构选定委员会对系统结构的评定，讲述评价系统结构方法中的二种，即评分法和典型程序测试法。

该委员会于 1975 年至 1976 年进行了对已有系列机系统结构的选优。由于计算机不断发展，计算机硬件和软件都在不断更新，所以该委员会的工作目标不是选择具体机器，而是选择适用于该委员会要求的系统结构。这里所指系统结构的定义，即我们在第一章讲过的，是从机器语言程序员所看到的计算机的属性。按这样定义的系统结构，可以做到选择机器系统结构与它如何具体实现无关，从而可适用于具体实现方法的不断改进，另外也便于采用模块化技术。对所选择的系统结构应能充分利用已有软件，也能适应将来设计的软件。

委员会先是初选九种系统结构，其型号为：B6700，IBM370，INTERDATA 8/32，GYK-12，PDP-11，ROLM1664，SEL32，UYK-7，UYK-20。接着，按委员会提出的一组定性和定量的性能判定标准用评分法进行研究，从上述九种筛选出三个，即 IBM370，PDP-11 和 INTERDATA 8/32。而后，用典型程序法等进一步选优，最后选定 PDP-11 的系统结构。

初选的九种系统结构是根据委员会向有关的 35 个单位征求意见后定下的。

2.6-1 评分法

委员会采用评分法进行予筛选。为了进行评分，委员会确定了九个必须满足的定性判定标准和十七个定量判定标准。九个定性标准是：

(1) 对虚拟存贮器系统的支持

所选系统结构必须是有虚、实地址转换机构，以支持虚拟存贮器系统的实现。在第五章已经详细地讲述了采用虚拟存贮器系统的好处。

(2) 保护机构

所选系统结构必须是有这样的硬件保护机构，它能在加进新的、正在调试的程序时，保证已有程序不会被破坏，并在出现某些灾难性软件故障时，进行保护隔离。在第五章也已谈到过保护结构。

(3) 浮点支持

所选系统结构必须明显地支持一种或多种具有 10 位十进制有效数字精度的浮点数据类型，这个有效数位是大多数实际应用所必需的。

(4) 中断和陷阱

对所选系统结构必须能编制出这样的陷阱处理程序，它能在响应和处理陷阱后恢复原来程序的执行。例如，处理机对地址转换部件的页面失效陷阱，必须能申请到相应的页面并把它从辅存调入，然后恢复原来的执行。如果恢复执行在逻辑上是不可能的（例如，企图向虚拟存贮器只读段存入一操作数等），则陷阱处理程序应当能中断这个程序，并给出不可能执行的指令或操作数的标记。

对中断也有类似的要求，结构必须能在任何中断（例如，电源故障、磁盘读错、控制台停机等）之后恢复执行。

不论是陷阱处理程序，还是中断处理程序应能运行于具有相同系统结构的各档机器。注意，这里所指的陷阱与我们在第四章所讲的有差别，它包括了那里讲的一些中断。

(5) 可分解性

所选系统结构应能使机器的组成部件中，至少有以下几种是能分解出来的：

- a. 虚、实地址转换机构；
- b. 浮点指令和寄存器（如果是与通用寄存器分开的）；
- c. 十进制指令系统；
- d. 保护机构。

满足这个要求，就可以使得同一系列机内的某些低档机可以不包括采用多道程序、具有多种用途的高档大型机的某些部分。

(6) 对多处理机支持

所选系统结构应该具有某些象第二章讲过的“测试和置定”形式的指令，以支持多个处理机之间的通讯和同步。

(7) 输入/输出的可控性

对所选系统结构，其处理机必须能行使对任何 I/O 处理机和 I/O 控制器的“绝对”控制。所选系统结构应能由处理机启动和询问到某个外设，然而，如何由处理机停止外设，却可以有各种办法，应该是既能停止所有输入/输出设备，也能停止指定的某个设备。

(8) 扩展性

所选系统结构必须有可能增加与已有指令格式一致的指令。要求至少有一个操作码码点是还没定义的。

(9) 只读编码

所选系统结构必须能执行在只读存贮器中的程序。这样，由于可使关键程序存在非易失

性的只读存储器，从而增加了系统的可靠性。只读存储器中的程序应该是可以改写的，即使这个结构没有专用类型的指令去实现这种改写，也应有其它方式可实现。

表 8.13 给出了按这些定性标准，对每个候选结构的评定结果。可以看出，九个候选结构对大部分定性标准都是满足的，这是因为初选时就有意地尽量靠近这些标准。但是，定性标准是很粗略的，没有精确定义，有些标准很难确切判定，对这些定性判定标准还应进一步仔细分析和下定义。

委员会规定的十七项定量标准如下：

虚拟地址空间

Y_1 ：虚拟编址空间的地址位数。

Y_2 ：虚拟编址空间按可编址单位计算的地址位数。

实地址空间

P_1 ：实地址空间的地址位数。

P_2 ：实地址空间按可编址单位计算的位数。

指令空间的未定义部分

为使所选择的结构有比预期的寿命还长的使用时间，在指令空间中应留有未定义的部分。设它所占指令空间的比例为 U ，则可按下式求得：

$$U = \sum_{1 \leq i < \infty} u_i 2^{-i}$$

其中 u_i 是长度为 i 位的未定义指令的条数。

中央处理机的状态数量

在第四章“中断系统”讲过，系统在处理中断或切换作业时，需要保存或加载的信息量对实时系统的响应时间以及多道程序系统的辅助操作时间有重要影响。处理机状态通常包括累加器、变址寄存器、指令计数器、条件码、存储器映象寄存器、中断屏蔽寄存器等的内容。

CS_1 ：完整结构的处理机状态位数。

CS_2 ：最小简化结构（即没有浮点、十进制、保护、地址变换寄存器）的处理机状态位数。

CM_1 ：完整结构在处理中断时，为保存处理机状态和中断完成时恢复原来处理机状态，需要在处理机与主存之间传送的位数。 CM_1 和 CS_1 不同，因为如果结构中有第四章“中断系统”讲过的可切换的寄存器组，那这些寄存器的内容就不必保存到主存中去。另外，控制执行状态保存所要用的指令也是由主存调出的，它们的总位数包括在 CM_1 ，而不包括在 CS_1 中。

CM_2 ：对应最小简化结构的 CM 值。

使用状况

B_1 ：至 1976 年 1 月为止交付使用的计算机台数。

B_2 ：至 1976 年 1 月为止已安装计算机的总值（单位为百万美元）。

这两项量度可以大致反映对应每种系统结构的已有软件状况和程序人员使用这种系统结构的经验积累。

表 8.13 候选结构按定性标准的评选结果

绝对标准	候 选 计 算 机 结 构									
	IBMS/370	Interdata 8/32	Rolm AN/UYK-28	DEC PDP-11	Univac AN/UYK-7	SEL 32	B-6700	Univac AN/UYK-20	Litton AN/GYK-12	
1. 虚拟存储器	Y	Y	N	Y	Y	N	Y	N	Y	
2. 保护	Y	Y	Y	Y	Y	Y?	N	N	Y?	
3. 浮点	Y	Y	Y	Y	N	Y	Y	Y	N	
4. 中断/陷阱	Y	?	Y	Y	Y	Y	Y	Y	Y	
5. 可分解性	Y	Y	Y	Y	Y?	Y	Y?	Y	Y?	
6. 多处理机	Y	Y	Y	Y	Y	Y	Y	Y	Y	
7. I/O 可控性	Y	Y	Y	Y	Y	Y	Y	Y	Y	
8. 扩展性	Y	Y	Y	Y	Y	Y	Y	Y	Y	
9. 只读编码	Y	Y	Y	Y	Y	Y	Y	Y	Y	
综 合	Y	?	N	Y	N	N	N	N	N	

(注) Y——满足标准

N——不满足标准

Y?——有条件地满足

?——没有结论

I/O 启动

I: 为了向标准外部设备发送一个 8 位字节需要在主存和中央处理机或 I/O 处理机之间传送的最少位数。

虽然想通过这项量度反映结构的响应性,然而,却难于对“需要传送的最少位数”下准确定义,因为这还和在输入、输出过程中需要执行哪些操作以及由何处取得参数等有关。所以,要从 I 值判定哪种结构的响应性好是不准确的。

可虚拟性

K: 如果结构是可以支持第一章提到过的虚拟机技术的, K 为 1, 否则 K 为 0。

判定这个标准想吸收虚拟机概念(如 IBM 的 VM/370)的优点。结构如果支持虚拟机,则它具有这样一种结构,能使一个特权独立程序作为一个非特权任务来运行,但产生的结果与作为特权程序运行时一样。这样,就可将一个操作系统以用户方式运行,从而成为另一个操作系统的子系统。

指令的直接编址能力

D: 一条指令可直接编址的最大主存位数(用所对应的地址位数表示)。

显然,这取决于指令中位移字段的长短;而它又随不同的指令格式而异。所以委员会规定这项量度是按标准的 LOAD(取)和 STORE(存)指令或与之等价的指令来定。例如 IBM 370 的 LOAD 指令,其位移字段为 12 位,因为它是字节编址,所以 $D = 15$ 。

最长的中断等待(时间)

L: 从一个中断发申请到 CPU,至该中断(没有被屏蔽的)开始被处理的这段时间内,在主存和任何处理机(中央处理机, I/O 控制器等)之间允许传送的最大位数。由这个标准可以判定系统中不可中断指令(或指令串)的最长指令周期。

子程序联结

J_1 : 在完整结构中,从转子到返回,包括保存用户状态,转到被调程序,恢复用户状态,返回调用程序,在处理机和存储器之间必须传送的位数,不包括参数的传送。

J_2 : 在最小简化结构中,与 J_1 对应的值。

这项标准可用于量度系列机中最高挡机和最低挡机的子程序调用,在最坏情况下所需的辅助操作量。

看出,由于所研究的是对系统结构的判定标准;因此,虽然它们是定量标准,但却不能和具体实现方法有关。所以,就不能有真正的时间量度,而是用处理机与主存间传送的位数来表征。

表 8.14 给出候选结构对这 17 项定量标准的量度结果。

下面研究应如何对测量结果进行评分。只有表 8.14 的定量标准原始数据,并不能直接看出哪个结构更好些。因此,必须对这些原始测量数据加以处理。

首先,这十七项定量标准并不是具有均等的重要性,所以应赋予它们不同的权值。委员会用投票方法决定了每个定量标准的权值,如表 8.15 所示。再者,各项标准的数值范围可能相差几个数量级,如果不加以处理,数值小的标准将在总评分中失去作用。为此应对原始数据加以归一化。归一化的方法是取每项标准对 9 台机器的平均值(如 I 的平均值为 68.6)作为数据的中心值取为 1。那么对 IBM370,其 I 值的归一值即为 $64/68.6 = 0.93$ 。

表 8.14 定 量 标 准 测 量 结 果

定量标准	IBM S/370	Interdata 8/32	Rolm AN/UYK- 28	DEC PDP-11	Univac AN/UYK -7	SEL 32	Burroughs B6700	Univac AN/UYK -20	Litton AN/GYK -12
1. V ₁	27	27	20	20	24	22	24	20	20
2. V ₂	27	27	20	19	24	22	20	17	20
3. P ₁	27	27	22	25	23	26	24	20	29
4. P ₂	27	27	22	24	23	26	20	17	29
5. U	.371	.355	.039	.043	.15	.450	.019	.125	.219
6. CS ₁	1344	1632	1008	1168	992	304	306	1328	1008
7. CS ₂	576	576	112	144	448	288	204	336	752
8. CM ₁	3168	1120	1882	736	1472	768	408	2256	1344
9. CM ₂	1312	32	544	480	1472	704	408	720	1088
10. K	1	0	0	1	0	0	0	0	0
11. B ₁	17.300	185	13.800	14700	346	75	90	400	30
12. B ₂	16.000	14	169	311	147	23	207	8	6
13. I	64	16	48	16	128	64	164	80	32
14. D	15	27	20	19	18	22	18	20	20
15. L	6192	560	114	112	2112	288	255	—	1376
16. J ₁	1904	2368	1360	1040	1280	960	459	1408	1344
17. J ₂	1136	1280	320	400	1280	960	459	640	1088

表 8.15 定量标准的权值

标准	V_1	V_2	P_1	P_2	U	CS_1	CS_2	CM_1	
数值	.0433	.0529	.0612	.0554	.0600	.0466	.0371	.0596	
标准	CM_2	K	B_1	B_2	I	D	L	J_1	J_2
数值	.0450	.0558	.0313	.0254	.1238	.1025	.0917	.0629	.0475

还有一个问题是有些项（如 V_1 , V_2 , P_1 , P_2 , U , K , B_1 , B_2 , D ）的数值是愈大愈好，有些项（如 CS_1 , CS_2 , CM_1 , CM_2 , I , L , J_1 , J_2 ）的数值则是愈小愈好。对数值愈小表明系统结构愈好的项，应取其数值的倒数。

经过归一化，取倒数和加权处理过的总评分结果示于表 8.16，它是按评分高低排列的。委员会据此筛选出 Interdata 8/32, PDP-11 和 IBM S/370 三种系统结构。

表 8.16 综合评分结果

结 构	评 分
Interdata 8/32	1.68
PDP-11	1.43
IBM S/370	1.36
AN/GYK-12	0.94
ROLM AN/UYK-28	0.92
B-6700	0.91
SEL-32	0.86
AN/UYK-7	0.46
AN/UYK-20	0.44

2.6-2 典型程序法

经过评分法筛选出的最后三种结构还需进一步对其性能进行分析。在前面讲过，评价计算机系统结构的实用方法中，最好的是典型程序法。典型程序测试法的核心问题是：

- (1) 如何选取一组具有代表性的测试程序；
 - (2) 在给定的人力条件下，程序员如何编写测试程序才能获得最准确、最有效的信息。
- 该委员会的具体做法如下：

1. 选用什么语言来编写程序

如果能用高级语言编写程序会有很多方便，但是没有一种实用办法能将编译的作用与系统结构的作用分开。而我们所研究的只是系统结构性能的量度，如混有编译程序的作用，就会掩盖了系统结构本身性能的好坏。用汇编语言编写程序也有一些明显的问题。首先，对每种结构的机器都要重写一次程序，因此增加了很多工作量。再者，由于程序员对不同系统结构熟练程度的不同，可能使编写的程序差别很大。还有，用汇编语言编写程序要比用高级语

言编写贵得多。但是委员会认为还是有办法限制和减少由于程序员不同所带来的影响，而却没办法分开或限制编译程序的影响。所以，最后还是决定用汇编语言编写程序。

2. 选择测试程序的指导原则

(1) 测试程序应该是尽可能少的核心类型的程序，每种测试程序的长度应不超过 200 条机器指令。这样能减少因程序员不同的影响，而且节省资源。

(2) 程序是用类似于 PL/I 的程序定义语言 PDL (Program Definition Language) 定义成结构式程序，然后用手编变为相应系统结构的汇编语言程序。

(3) 不允许程序员改换算法，只能将 PDL 描述的转为汇编语言。使手编程序受限于严格规定的算法，是希望减少由于程序员熟练程度不同而产生的影响。

(4) 除了 I/O 中断测试程序外，所有测试程序都是可再入编码并采用和位置无关（可浮动的）子程序。

以上这些，都是与当前好的程序设计技术相一致的。另外也能由此反映系统结构的子程序和编址性能。

3. 十二个测试程序的选择

委员会从有广泛代表性的 21 个测试程序中选取了 12 个。它们是：

(1) 具有 4 个优先级的 I/O 核心程序。这个程序使处理机能处理由不同优先级的四个设备发来的中断，并且当某个设备的中断申请正在被处理时，允许响应优先级更高的设备发来的中断。

(2) 先来先服务的 I/O 核心程序。这个程序使处理机按先来先服务的排队次序处理由四个设备发来的中断，不考虑它们的优先级。当某设备的中断申请正在被处理时，别的设备发来的中断都可被响应。

(3) I/O 设备处理程序。用于处理应用程序要求的磁带 I/O 成组传送，在传送完毕应有状态回答。

(4) 大型快速福里哀变换程序。它计算 32 位浮点复数向量的快速福里哀变换。这个典型程序并不是用于测试机器的浮点运算能力，而主要是用于测试对大地址空间的管理能力（这个向量需要的字节数最多可达 0.5M 字节）。

(5) 位测试与置位或复位程序。它测试位串中某一位的初值，然后置位或复位。这是对机器位操作能力的测试。

(6) 隆格—库塔积分程序。它对一个简单微分方程用三重隆格—库塔法求解。这主要用于测试浮点运算和循环控制。

(7) 链表插入程序。在双向链表中插入一个新项。这个程序用于测试对指针型数据的操作能力。

(8) 快速分类。用快速分类算法对一个足够大的定长向量串进行分类。它也用于测试对大地址空间的管理能力以及机器对递归程序的支持。

(9) ASCII 码到浮点数的变换程序。这个程序测试机器从字符到数值的变换能力。

(10) 布尔矩阵转置程序。它转置一个紧凑存贮的方阵。用于测试以任意增量顺序经过位向量的能力。

(11) 虚拟存储器空间交换程序。这个程序改变处理机虚拟存储器的映象。

(12) 字符查找程序。从一个长字符串中顺序查找第一个出现的给定的自变量串，它能

反映顺序通过字符串的能力。

本来，通过这些测试程序是能够对不同机器进行比较，但我们现在所要进行的是和实现方法无关的不同系统结构比较，因此就不能用测试程序的真正执行时间来量度。然而评价计算机中，最感兴趣的还是空间和时间量度。为此，委员会定义了以下几个量度标准：

空间的量度：

S：该系统结构对此测试程序所需的字节数。

时间的量度：

M：在执行测试程序时，需要在主存和处理机之间传送的字节数。

R：在执行测试程序时，需要在处理机内各寄存器之间传送的字节数。

同一种测试程序对不同的系统结构所需占用的主存空间不同，这是量度系统结构的重要指标，选定 S 量度就是为此目的。它包括程序的所有指令、间接地址和执行程序所需的临时工作区，但程序用的数据结构或参数不包括在内。

大家都习惯于用运算速度来衡量机器的好坏，然而，这是和具体实现技术密切相关的。对于系统结构来讲，我们要求定义出能影响速度状况的量度标准。大家知道，中央处理器和主存间的频宽是反映计算机速度的一个重要标志。它并不涉及主存和处理机的内部状况，而只决定于它们之间的总线宽度和这个总线在单位时间（每秒钟）内能传送的字节数。因此，我们就可以用一个测试程序在执行时需在主存和处理机间传送的字节数来量度系统结构所反映的速度特征。譬如，为了执行一个测试程序，某个系统结构由主存读或写 2×10^6 字节，而另一个系统结构为了执行这同一个测试程序只需要由主存读或写 10^6 字节，那么，对于相同的实现技术，我们可以预计后者的执行速度一定比前者快得多。这就是委员会选定 M 量度的

表 8.17 IBM370 求内积程序的 M 量度

程 序 编 码	M	注 释
(1) LA 2,10(0,0)	4	置 $R_2 = 10$ 向量长度
(2) LA 3,XVEC	4	将 X 向量的开始地址加载 R_3
(3) LA 4,YVEC	4	将 Y 向量的开始地址加载 R_4
(4) SDR 2,2	2	清除浮点寄存器 2
(5) SR 7,7	2	清除 R_7 ，用它作指向浮点向量的变址
	16	以上 M 值合计
(6) LOOP LE 4,0(7,3)	8	加载 $X(i)$ 到浮点寄存器 4
(7) ME 4,0(7,4)	8	乘 $X(i)$ 与 $Y(i)$
(8) ADR 2,4	2	$SUM := SUM + X(i) * Y(i)$
(9) LA 7,4(0,7)	4	变址地址加 4 字节
(10) BCT 2,LOOP	4	循环计数减，若没做完返回循环
	26	循环一次 M 值
	260	(6) 至 (10) 循环 10 次 M 值
(11) STO 2 SUM	12	存双精度结果在 SUM
	288	M 值总计值

意义。然，如果采用 Cache 存储器或指令预处理缓冲器等技术是会减少处理机和主存间的所需送的信息量。但是，为了量度的简单、明瞭和易行，委员会不考虑这些。

从器语言程序是可以算出M值。表 8.17 给出 IBM370 求内积的一段程序的M值的计算结果

当，只有M的量度还不足以完全反映出系统结构对速度的影响，中央处理机内部寄存器间的数传送流量 R，也是一个重要量度。委员会用M和 R 两个量度的合成结果($aM + bR$)来衡量处理机的速度状况，其中 a、b 是适当选定的系数。为了简化 R 的测量，认为处理机的内部结构是理想、规整的数据通路，运算逻辑部件只对整数、浮点数和十进制数进行算术运算、加“1”、减“1”和移位或循环移位。R 值中只包括数据流量，不包括控制流量。

关于 R 量度的定义是委员会讨论的重要核心问题。并定出了一整套规则。在每条指令的操作序列及所用通路确定后，就可以算出每种指令的 R 值。例如，执行一条 IBM370 寄存器型加法指令，需要在处理机内各寄存器之间传送 49 个字节。由每条指令的 R 值就可求出每个典型程序的 R 值。

委员会对筛选出的三种系统结构按 12 种测试程序算出其 S、M和R值，共用了16名程序员。对所有组合情况的计算都进行，就要做 576 次。显然，时间、经费都不允许这样做。他们用了数理统计的方法，进行细致的数据采集和整理工作。最后得到的结果示于表8.18。从表 18 可看出，对同样的测试程序，IBM S/370 需要比 PDP—11 多21%的主存容量。虽然从 S、M、R 的量度来看，Inter data 8/32 的结构最佳，其次是 PDP—11 的，而 IBM5/370 的最差。但综合其它各种性能指标，委员会选定了 PDP—11 的系统结构。

表 8.18 对12种测试程序的 S、M、R 值

结 构	S	M	R
PDP—11	1.00	0.93	0.94
IBM S/370	1.21	1.27	1.29
Interdata 8/32	0.83	0.85	0.83

应该指出，委员会于77年发表的这份评定报告只是对系统结构的一种评价方法。我们在这里并不是着眼于论述哪种系统结构最好，而是想通过这一具体例子中使用的评分法和典型程序，使大家对于应该从哪些方面来评价系统结构有些具体的认识。另外，这两种方法中应用了前述第一章到第五章中的一些重要概念，因此可以说，这一节是这些重要概念的具体应用和深化。

主要参考文献

- [1] D. Ferrari, "Computer Systems Performance Evaluation," Prentice-Hall, 1978.
- [2] 陈天雄 (加拿大, 哥伦比亚大学), "计算机系统性能评价", 在西北电讯工程学院讲学讲稿, 1980.
- [3] L. Svobodova, "Computer Performance Measurement and Evaluation Methods,

Analysis and Applications," Elsevier, 1976.

- [4] S.H. Fuller, "Performance Evaluation," Ch. 11 in "Introduction to Computer Architecture," ed., H. Stone, Second Edition, Science Research Associates, 80.
- [5] Yaohan Chu, "Users' Manual for the CDL3 Simulator," University of Maryland, May 1978.
- [6] M. E. Drummond, Jr., "Evaluation and Measurement Techniques for Digital Computer Systems," Prentice-Hall, 1973.
- [7] Stephen Y. H. Su, "Computer Hardware Description Languages and Their Application, an Introduction and Prognosis," Computer, Vol. 10, No. 6, June 1977, pp. 10-13.
- [8] Yaohan Chu, et al, "Why Do We Need Computer Hardware Description Languages?" Computer, Vol. 7, No. 12, Dec. 1974, pp. 18-22.
- [9] G. J. Lipovski, "Hardware Description Languages, Voices from the Tower and Babel," Computer, Vol. 10, No. 6, June 1977, pp. 14-17.
- [10] Yaohan Chu, "Introducing CDL," Computer, Vol. 7, No. 12, Dec. 1974, pp. 31-33.
- [11] S. S. Ching and J. H. Tracey, "An Interactive Computer Graphics Language for the Design and Simulation of Digital Systems," Computer, Vol. 10, No. 6, June 1977, pp. 35-41.
- [12] W. E. Burr, et al, "Overview of the Military Computer Family Architecture Selection," AFIPS Conf. Proc., Vol. 46, 1977, pp. 131-137.
- [13] S.H. Fuller, et al, "Initial Selection and Screening of the CFA Candidate Computer Architectures," AFIPS Conf. Proc., Vol. 46, 1977, pp. 139-146.
- [14] S. H. Fuller, et al, "Evaluation of Computer Architectures via Test Programs," AFIPS Conf. Proc., Vol. 46, 1977, pp. 147-160.
- [15] C.C. Easter, et al, "Measures of Op-Code Utilization," IEEE Trans. on Computers, Vol. C-20, 1971, pp. 582-584.